

Windows

游戏编程大师技巧

2D 和 3D 游戏编程基础



ANDRE LAMOTHE 著
曲文卿 姚君山 钟澍莹等 译

SAMS

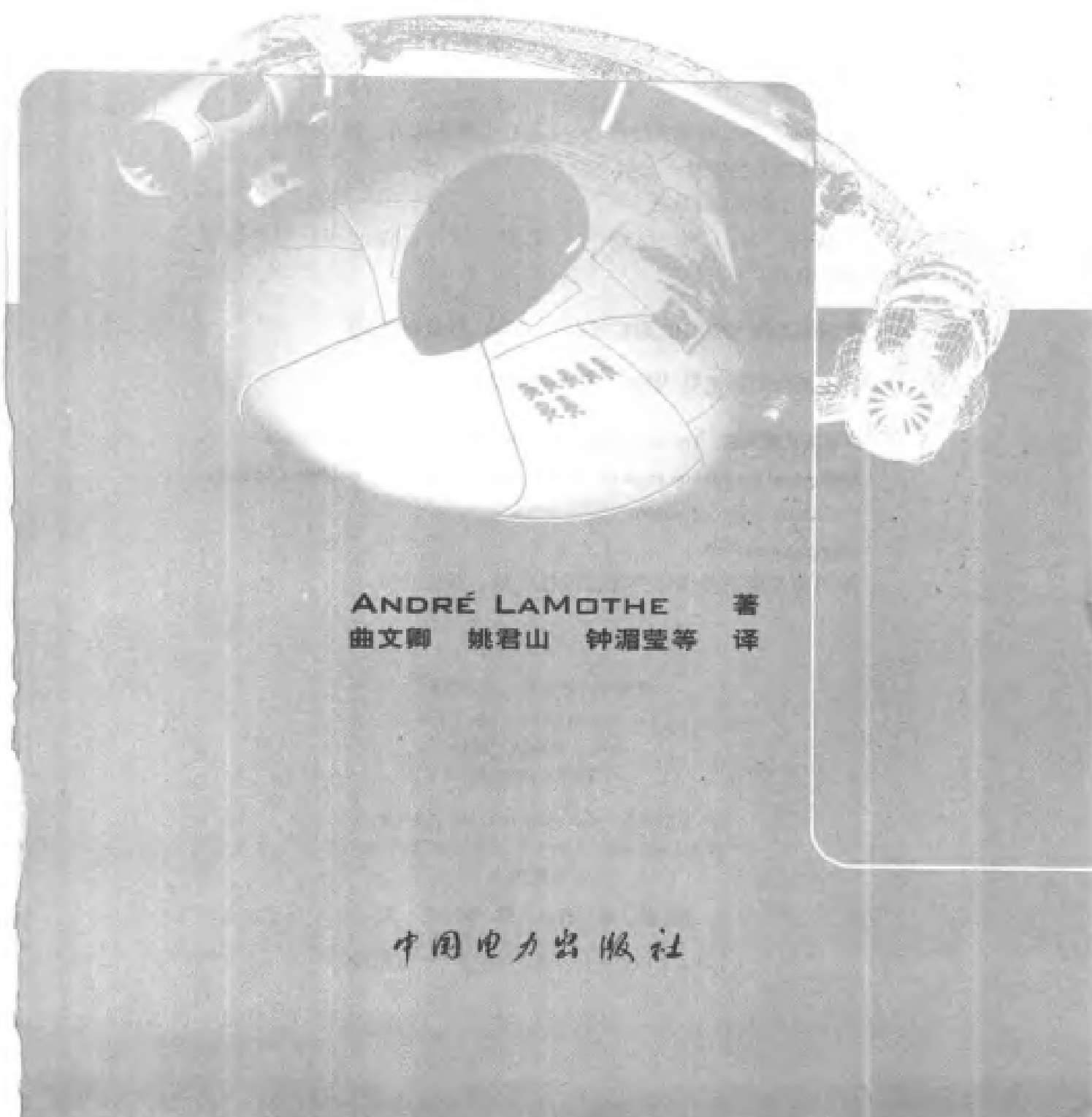


中国电力出版社
www.infopower.com.cn

Windows

游戏编程大师技巧

2D 和 3D 游戏编程基础



ANDRÉ LAMOTHE 著
曲文卿 姚君山 钟涓莹等 译

中国电力出版社

内 容 提 要

本书介绍了在 Windows 环境下进行游戏编程所需用到的各方面知识。全书正文共分为一个部分、十四章。第一部分为 Windows 编程基础,包括概述、Windows 编程、GDI、控件等知识,第二部分为 DirectX 和 2D 基础,接触了用 DirectX 进行各种控制及二维平面中变换的知识,第三部分编程核心则重点介绍了相关的数学、物理原理及如何进行综合运用,另外第四部分附录提供了光盘简介、C/C++编译器、数学回顾、C++基础、游戏编程资源及 ASCII 表,为学习提供了方便。

本书适合有一定数学基础和 C 语言编程经验的读者阅读。

图书在版编目 (CIP) 数据

MS 526/05

Windows 游戏编程大师技巧/ (美) 拉莫斯编著; 曲文卿等译. -北京: 中国电力出版社, 2001

ISBN 7-5083-0734-8

I. W... II. ①拉...②曲... III. 窗口软件, Windows-程序设计 IV. TP316.7

中国版本图书馆 CIP 数据核字 (2001) 第 066979 号

著作权合同登记号 图字: 01-2000-3335 号

本书英文版原名: Tricks Of The Windows Game Programming Gurus

Authorized translation from the English language edition, published by Sams

Publishing Copyright©2001

All rights reserved.

本书中文版由美国培生集团授权出版, 版权所有。

中国电力出版社出版、发行

(北京三里河路 6 号 100044 <http://www.infopower.com.cn>)

实验小学印刷厂印刷

各地新华书店经售

*

2001 年 11 月第一版 2001 年 11 月北京第一次印刷

787 毫米×1092 毫米 16 开本 55.5 印张 1261 千字

定价 89.00 元

版 权 所 有 翻 印 必 究

(本书如有印装质量问题, 我社发行部负责退换)

序

我记得第一次和计算机结缘是在 1993 年，那时使用 Apple IIe 计算机编写 Logo 语言（感谢 Woz!）。那次经历对我而言是那样的令人迷醉，并且深深地印在我的脑海中。计算机可以做我所有想做的事，并且经过无数次循环都不疲劳。我的大量的工作都得益于那次经历，Andre Lamothe 设计的电影“War Games”。

我购买的 Andre Lamothe 的第一本书是在 1994 年：《Sams Teach Yourself Game Programming in 21 Days》。在此之前我从来都不认为人们可以将编写视频游戏作为一个职业。就是那时，我看到了喜爱编程和沉迷于视频游戏的结合点。以前有谁能够认同花大量的时间玩 Galaga 游戏在现在来说是研究呢？Andre 的撰写和教授风格鼓舞了我，给了我巨大的信心——我也能够编写视频游戏的程序。我还记得我曾在电话中向他请教，那是基于他的气体模型演示程序的物理作业的简单的程序（我至今还不能相信他会和任何一个打电话给他的人交流，并将他的电话号码给任何人）。我的程序无论如何都不运行，他立即检查了我的程序，并且在几秒钟后说：“Rich，你应当在每一行的后面添上分号！”就是这样简单，我完成了我的第一个游戏程序。

几年后我有幸作为一个工具程序员和 Andre 先生一起制作了一个视频游戏 Rex Blade。对我来讲那是一次很好的学习机会。我们在一起极其努力地工作（Andre 先生是一个苛刻的上司），但是充满了乐趣（看电影、射击、滑雪等等），最后完成了一个 3D 交互式视频游戏三部曲。在不可思议的短短 6 个月的时间中，我们就完成了 Rex Blade 游戏从开始设想到上架的全过程（Rex 进行了有趣的事后检验）。编写 Rex 使我懂得了如何制作一个真正的视频游戏，和 Andre 先生共事使我了解了什么是真正的夜以继日地工作，我说的是真正的昼夜不停地工作。而以前当我听说他一周工作 100 个小时的时候，还以为他在说谎。

很少有软件工程领域对硬件、软件和程序员本身的限制像游戏编程一样多。将数学、物理、人工智能、图像、声音、音乐、GUI 和数据结构等等内容完美地结合在一起是非常复杂的。这就是《Windows 游戏编程大师技巧》在目前和将来的视频游戏编程艺术中得以成为一种重要工具的关键所在。

本书将使你在游戏编程的技术上提高一个档次。仅仅是人工智能部分内容就能使你深感有趣，而演示程序更让人留连忘返。另外还可以学习到模糊逻辑、神经网络和遗传算法以及如何将上述内容应用到视频游戏中的内容。通过本书还可以学习到 DirectX 的所有主要组件，包括 DirectDraw、DirectInput（以及力反馈内容）、DirectSound 以及 DirectMusic 的最新、最重要的技术。

本书还介绍了物理建模方面的内容。最后，有余力的人还可以学习一些完全的碰撞反应、动量传递、正向运动以及如何实时模拟等内容。我们不难想像，能够学习的生物、能

够在类似实际的情况下发生碰撞的物体，以及能够记住在最后一次是如何被你击败的敌人等这些内容，都将是未来制作大型游戏的基础。

我不得不举手赞成 Andre 先生撰写这本书。他也一直说，如果他不做，那谁来做呢？的确是这样的：将一个人的 20 多年的艰苦工作、秘密和技巧奉献给每一个人是一件多么崇高的事情啊！

随着技术不断跳跃式地发展，我认为已经到了一个技术活跃的重要时刻了，特别是对于一个游戏程序员来说更是这样。每几个月就要推出一种新型的 CPU、视频卡和其他的硬件设备，将技术向前推进。（Voodoo III 每秒钟运行 70 亿次的速度几乎难以置信。）这种伟大的技术伴随着价格也同样昂贵。随之而来的就是期待我们制作的游戏能够使用该技术，这将减少未来视频游戏的障碍。这一切似乎在不远的将来就可以实现。惟一的限制因素就是我们的知识和想像力。

令我激动的是，下一代游戏程序员将得到本书的鼓舞和指导。我认为 Andre 先生也希望在 21 世纪有人继续他的工作，继续散布这项瑰丽的艺术，因为 Andre 先生也需要休息。

Richard Benson
3D Game Programmer
Dream Works Interactive

关于作者

Andre LaMothe 具有 22 年多的编程经验，拥有数学、计算机科学和电子工程的学位。他写了大量关于图形、游戏编程和人工智能方面的论文。他是本书、《Sams Teach Yourself Game Programming in 21 Days》、《The Game Programming Starter Kit》、《The Black Art of 3D Game Programming》以及《Windows Game Programming for Dummies》等所有畅销书的作者，另外还是《Ciarcia's Circuit Cellar I and II》的合作者。LaMothe 先生还曾在圣克鲁兹大学扩展多媒体系担任教学工作。

最后，Andre 是 Xtreme Games LLC 公司的创始人和执行总裁，Xtreme Games LLC 公司是一个智囊团和世界上最大的虚拟游戏公司，由 250 多个独立的开发工作室构成。

Andre 先生的联系方式：ceo@xgames3d.com。

感谢数字格式在线书籍的作者

在 CD 上的位置：T3DGAME\ONLINEBOOK

Matthew Ellis: 《Direct3D Primer》的作者

Matthew 是一个有着 10 多年 3D 游戏编程经验的程序员及作者，他居住在拉斯维加斯，对 3D 游戏编程和图形的各个方面都有兴趣。他目前正在创制一个新的 3D 引擎，同时还发表论文并自己编写书籍。

联系方式：matt@magmagames.com。

Sergei Savchenko: 《General 3D Graphic》的作者

Sergei 是位于 Montreal 的 McGill 大学计算机科学专业的一名毕业生。他来自于前苏联乌克兰的 Kharkov 城 (XAPbKOB)。

除了计算机科学专业的学习之外，Sergei 还在 Kharkov 航空研究所学习过飞行器设计专业。他也担任着计算机科学专业的授课工作，并且积极进行自动推理方面的研究工作。

联系方式：savs@cs.mcgill.ca。

网页：<http://www.cs.mcgill.ca/~savs/3dgp1/>。

David Dougher: 《Genesis 3D Engine Reference, Tool, and API Function Manuals》的作者

David 已经有 25 年以上的编程和游戏经验，1974 年在 Syracuse 大学就在 PDP-8 系统上使用纸带制作了他的第一个计算机游戏。他对游戏杂志的收藏要追溯到《Strategic Review》（攻略述评）杂志第一版（《Dragon》杂志的前身）。David 目前在 Parlance 公司担任全职发行工程师，喜欢 Babylon 5、Myst、Riven、Obsidian 以及游戏设计，另外还和他的妻子一起在舞厅教跳舞。

联系方式：ddougher@ids.net。

感谢文章和论文的作者

在 CD 上的位置: T3DGAME\ARTICLES

Bernt Habermeier: 《Internet Based Client/Server Network Traffic Reduction》的作者, Email: bert@bolt.com, 主页: <http://www.bolt.com>。

Ivan Pocina: 《KD Trees》的作者, Email: ipocina@aol.com。

Nathan Papke: 《Artificial Intelligence Voice Recognition and Beyond》的作者, Email: nathan.papke@juno.com。

Semion S.Bezrukov: 《Linking Up with DirectPlay》的作者, Email: deltree@rocket.mail.com。

Michael Tanczos: 《The Art of Modeling Lens Flares》的作者, Email: webmaster@logic-gate.com。

David Filip: 《Multimedia Musical Content Fundamentals》的作者, Email: grimlock@u.washington.edu。

Terje Mathisen: 《Penium Secrets》的作者, Email: terjem@hda.hydro.com。

Greg Pisanich 和 Michelle Prevost: 《Representing Artificial Personalities》和《Representing Human Characters in Interactive Games》的作者, Email: gp@garlic.com和prevost@sgi.com。

Zach Mortensen: 《Polygon Sorting Algorithms》的作者, Email: mortensl@nersc.gov。

James P.Abbott: 《Web Games on a Shoestring》的作者, Email: jabbott@longshot.com, 主页: <http://www.longshot.com>。

Mike Schmit: 《Optimizing Code with MMX Technology》的作者, Email: mschmit@zoran.com、mschmit@ix.netcom.com。

Alisa J.Baker: 《Into the Grey Zone and Beyond》的作者, Email: abaker@gcounsel.com。

Dan Royer: 《3D Technical Article Series》的作者, Email: aggravated@bigfoot.com, 主页: <http://members.home.com/droyer/index.html>。

Tom Hammersley: 《Viewing Systems for 3D Engines》的作者, Email: tomh@globalnet.co.uk。

Bruce Wilcox: 《Applied AI: Chess is Easy. Go is Hard》的作者, Email: brucewilcox@bigfoot.com。

Nathan Davies: 《Transparency in D3D Immediate Mode》的作者, Email: alamar@cable.net。

Bob Bates: 《Designing the Puzzle》的作者, Email: bbates@legendent.com。

Marcus Fisher: 《Dynamic 3D Animation Though Traditional Animation Techniques》的作者, Email: mfisher@avalanchesoftware.com。

Lorenzo Phillips: 《Game Development Methodology for Small Development Teams》的作者, Email: pain19@ix.netcom.com。

Jason McIntosh: 《Tile Graphics Techniques 1.0》的作者。

另外, 该 CD 上还选取了游戏编程杂志网站上的一些文章, <http://www.perplexed.xom/>。作者分别是: *Matt Reiferson、*Geoff Howland、Mark Baldwin、John De Goes、*Jeff Weeks、Mirek、*Tom Hammersley、Jesse Aronson、Matthias Holitzer、Chris Palmer、Dominic Fillion、JiiQ、Dhonn Lushine、David Brebner、Travis “Razorblade” Bemanm、Jonathan Mak、Justin Hust、Steve King、Michael Bacarella II、Seumas McNally、Robin Ward、Dominic Fillion、Dragun、Lynch Hung、Martin Weiner、Jon Wise 和 Francois Dominic Laramé。

*表示有多篇论文。

谨以此书献给我真挚的同事 Jennifer Jane

致谢

我一直不喜欢写致谢，因为完成一本书，需要太多的人参与，无法对每一个人都表示感谢。然而，现在我还是要对所有为完成本书做出贡献的人和公司表示我的谢意，致谢不分任何次序。

首先要感谢我的父母，在他们晚年时养育了我，给了我这么多的与众不同之处，不需要休息，可以不间断的连续工作。谢谢父母！

其次，我想感谢 Macmillan 计算机出版社的所有工作人员，他们为我以自己的风格撰写本书提供了帮助。让美国人做超出常规的事的确是一个考验，而我不是一个墨守成规的人，但是要想打开一个新局面，就必须这样做。我要特别感谢采稿编辑 Angela Kozlowski，他听取了我的艺术/市场的概念，并将它们付诸实现；感谢项目编辑 Carol Bowers 肯定并且采用了我的“编辑越少越好”的观点；感谢媒体和权限经理 Dan Scherf 确定将所有的程序制作成 CD；还要感谢开发编辑 Erik Dafforn 保证了上百个图以及 1000 余页的手稿没有损坏。

当然还要感谢为本书工作的所有的其他编辑和排版人员，包括 Steven Haines、Sean Medlock、Carol Ackerman、Kezia Endsley 和 Howard Jones。看上去他们在本书编辑过程中只是进行了一些幕后的工作，但是他们极好地完成了工作。特别感谢 Steve 和 Sean，他们为我检查出了许多愚蠢的错误。

还有，我要感谢 Microsoft 的 DirectX 公司，特别是要感谢 Kevin Bachus，它提供了最新版本的 DirectX SDK 的内容，并且肯定了我的所有主要的 DirectX 部分的内容。这些内容对我非常重要，非常感谢！

我还要感谢为本书提供软件或其他重要内容的所有公司。主要有：Caligari 公司提供了 TrueSpace 的使用，JASC 提供了 Paint Shop Pro 的使用，以及 Sonic Foundry 提供了 Sound Forge 的使用。另外还要感谢 Matrox 和 Diamond Multimedia 提供了 3D 加速器的演示程序，Creative 实验室提供了声卡，Intel 公司提供了 VTune，Kenitics 提供了 3D Studio Max，Microsoft 公司和 Borland 公司提供了编译器产品等。

还要感谢在这艰苦的编写过程中和我紧密联系的所有朋友。感谢 Gold's Gym 的所有的人：Armand、Andrew、Paul 和 Dave。感谢 Mike Perone，随时检查出软件的难于发现的错误。还要感谢我的朋友 Mark Bell——我一直怀念他，Happy 先生——从 8 年前一起滑雪旅行至今还欠我 180 美元！（我不能再忍受了；请快一点，Mark，我一直在等待你归还我的钱）。

我还要感谢的是提供 CD 上的相关论文的所有作者。如果没有你们的话，可怜的读者将只能阅读到我的片面的一家之言。要特别感谢 CD 上 Direct3D 书籍的作者 Matthew Ellis，以及为本书题写序言的 Richard Benson(Keebler)先生。

最后，我要感谢每天和我在一起的、一直给我帮助的、我的朋友 Jennifer。

感谢所有的人！

请将您的想法告诉我们

作为本书的读者，您是我们最重要的批评者和注释者。我们非常重视您的看法，并且希望了解正确的、更好的方式、您更喜欢什么内容的出版物以及您想通过我们传递的建议。

作为 Sams 的出版商来讲，对您的意见我们非常欢迎。您可以通过传真、电子邮件或写信来告诉我们，您是否喜欢该书的内容，以及我们应当如何改进才能使得我们出版的书籍内容更加充实。

请注意我们不能帮助您解决有关本书内容的技术问题，这主要是由于本人收到的信件过多，以至于不能每封信都答复。

如果您给我写信的话，请确认填写好本书的主题和作者，以及您的姓名、电话或传真号码。我将认真地阅读您的问题，并将它们转交给本书中相关的作者或编者。

传真: (317)581-4770
Email: mstephens@mcp.com
邮寄地址: Michael Stephens
Publisher
Sams
201 West 103rd Street
Indianapolis, IN 46290 USA

简 介

“无论生与死，你都将和我在一起。”

—Robocop

很久很久以前，在遥远的星系，我撰写了一本关于游戏编程的书，名字是“Tricks of the Game Programming Gurus”。对我而言，它只是为我提供了写一本教读者如何制作游戏的一个机会。很多年来，我就一直有这个念头。我的岁数要大一些，学到的东西也多一些，并且确实学到了许多技巧。本书也弥补了一些以前出版的书遗漏的问题。我将主要讨论束缚游戏编程的每一个主要的问题！

但是和往常一样，我认为你还不是一个编程专家，甚至不知道如何制作游戏。本书对于初学者和高级游戏程序员同样适用。尽管如此，本书讲述的速度还是很快的，所以千万不要稍有懈怠。

现在可能是历史上游戏编程行业的最好的时代。我的意思是现在我们拥有了制作逼真的游戏的技术。想像一下下一步会发生什么呢？不过这些技术并不容易掌握，需要进行刻苦的学习。近来在制作游戏的技巧方面确实出现了一些障碍，但是如果你阅读了本书，你可能就能成为一个喜欢挑战障碍的人，不是吗？好了，现在你已经准备就绪了，因为当你开始利用本书内容时，就能够制作一个全 3D 的、有组织计划的、专业水准的 PC 机上运行的视频游戏，并且能够理解有关人工智能、物理建模、游戏算法和 2D/3D 图像的基本理论，使用当前和未来的 3D 硬件。

所学的内容

通过本书可以掌握 100 多个信息！我将以大量的你可能出现遗漏的信息填充你的大脑。尽管这样说夸张了点，但是本书内容确实包括了制作 PC 计算机上基于 Windows 9X/NT 系统的游戏的所有必须的内容：

- Win32 编程
- DirectX 基础
- 2D 图形和算法
- 游戏编程技术和数据结构
- 多线程编程
- 人工智能
- 物理建模

- 使用 3D 加速硬件设备（在 CD 上）

还有更多的内容……

本书内容主要集中在游戏编程方面。CD 上有两套计算机书籍，分别包含了 Direct3D 快速方式和通用 3D 方面的内容。

应当具备的基础知识

首先你应当是一个程序员，才能阅读本书内容。如果不会编写 C 程序的话，对你将有很大的影响。本书还使用了一些 C++ 语言，只能编写 C 的程序员可能会感到稍有一些难度。当然如果在一些稍微古怪的地方，我会提示读者；并且在附录 D 中我附加了一些关于 C++ 的基础内容，以便于在你紧急需要时可以阅读一下。基本上来讲，本书中只需要 C++ 的知识，例如在使用 DirectX 的时候。

毋庸置疑，我在本书中使用了较多 C++ 的知识，因为在游戏编程过程中有许多问题都需要使用面向对象的技术，而如果强迫使用类 C 语言的结构，就会出现画虎不成反类犬的问题。最低限度是，能够使用 C 程序就可以阅读本书了。如果能够使用 C/C++ 来编程的话，那就根本没有任何问题。

众所周知，一个计算机程序只是一个逻辑和数学的过程。而 3D 视频游戏重点在于数学部分！3D 图形全部都是数学模型。幸运的是那只是比较容易的数学内容（当然数学本身就并不很容易）。只要了解一些基本的代数和几何知识就可以了。同时我将顺便教授一部分的矢量和矩阵的理论。只要会一点矢量（矩阵）的加、减、乘、除运算，就可以理解 90% 的内容，而不必从头开始学习这些内容。只要能够应用这些程序，那一切就都解决了。

上面内容就是所有的你应当具备的基础知识。当然，你最好告诉所有的朋友，在大约两年之内他们都将看不到你，因为在这段时间中，你将非常忙碌。但是在你学会制作游戏程序后，就可以尽情享受了！

本书的组织

《Windows 游戏编程大师技巧》分为 4 大部分，14 章，6 个附录。

第一部分 Windows 编程基础

- 第一章 无尽之旅
- 第二章 Windows 编程模型
- 第三章 高级 Windows 编程
- 第四章 Windows GDI、控件和突发奇想

第二部分 DirectX 和 2D 基础

- 第五章 DirectX 基础和令人生畏的 COM
- 第六章 首次接触：DirectDraw
- 第七章 高级 DirectDraw 和位图图形
- 第八章 矢量光栅化及 2D 转换
- 第九章 利用 DirectInput 和力反馈进行输入
- 第十章 用 DirectSound 和 DirectMusic 演奏乐曲

第三部分 编程核心

- 第十一章 算法、数据结构、内存管理及多线程
- 第十二章 人工智能在游戏中的运用
- 第十三章 基本物理建模
- 第十四章 综合运用

第四部分 附录

- 附录 A CD 上的内容
- 附录 B 安装 DirectX 和使用 C/C++ 编译器
- 附录 C 三角函数和矢量
- 附录 D C++ 基础
- 附录 E 游戏编程资源
- 附录 F ASCII 表

安装 CD-ROM

CD-ROM 上含有全部的源程序代码、可执行程序、实例程序、存储技巧、3D 建模程序、音效以及补充本书的辅助技术文章。目录结构如下：

```
CD-DRIVER:\
T3DGAME\
SOURCE\
    T3DCHAP01\
    T3DCHAP02\

    T3DCHAP14\
APPLICATIONS\
ARTWORK\
    BITMAPS\
    MODELS\
SOUND\
    WAVES\
    MIDI\
DIRECTX\
GAMES\
GOODIES\
ARTICLES\
ONLINEBOOKS\
ENGINES\
```

每一个主目录下含有所需要的技术数据。下面是更详细的细目分类：

T3DGAME——含有所有其他目录的根目录。在每次更换目录前，请首先阅读一下 README.TXT 文件。

SOURCE——含有本书所有的按章节排列的源目录。只要将整个 SOURCE\ 目录全部复制到硬盘上，就可以从硬盘上运行。

DEMOS——含有许多公司授权我使用的演示程序。

ARTWORK——包含可以在你的程序中免费使用的存储原图。

SOUND——包含可以在你的程序中免费使用的存储的音效和音乐。

DIRECTX——包含最新版本的 DirectX SDK。

GAMES——包含大量的 2D 和 3D 的非常不错的共享软件游戏。

ARTICLES——含有游戏编程界中的大师们撰写的利于启发游戏编程灵感的论文。

ONLINEBOOKS——含有两套完整的数字在线手册，分别包含了 Direct3D 快速方式和通用 3D 图形。

ENGINES——含有大量的 3D 引擎。

该 CD 由于包含许多不同类型的程序和数据，因而没有通用安装程序。你需要自己安装。但是，在大多数情况下，只要将 SOURCE\目录复制到硬盘上，在硬盘上运行就可以了。对于其他程序和数据，当你需要它们时，再安装这些程序和数据。只要将它们拖到硬盘上，在各自的目录中运行各自的安装程序即可。

安装 DirectX

该 CD 中必须安装的最重要的部分是 DirectX SDK 和运行时间文件。安装程序在 DIRECTX\目录下，该目录下还有 README.TXT 文件的解释和最新变动。

注意



你必须安装 DirectX6.0SDK 及以上版本以运行光盘。如果你不清楚你的系统中是否有较新的文件，可运行安装程序，它将会提示你。

编译程序

我使用了 Microsoft Visual C++5.0/6.0 来编写本书的程序。但是大多数情况下，这些程序可以使用任何一种 Win32 兼容的编译器来进行编译运行。毫无疑问，我建议使用 Microsoft VC++，因为它的运行状况最好。

如果你不熟悉编译器的 IDE，在编译 Windows 程序时将非常麻烦。请花一点时间学习一些编译器的知识，在编译程序之前，至少应当了解如何编译一个“世界你好”的控制程序或者其他类似的基本程序。

要编译 Windows Win32.EXE 程序，应当将设置应用程序工程的目标程序设定为 Win32.EXE，然后再进行编译。但是要创建 DirectX 程序的话，必须在工程中包含 DirectX 输入库函数。如果认为只要将 DirectX 库函数添加到你的 include 路径中就可以了的话，那这样的结构根本不能运行。不要给自己添乱了，将 DirectX.LIB 文件手动包含到工程中或者是工作空间中。在安装的 DirectX SDK 主目录下的 LIB\目录下可以找到.LIB 文件。这样就不会导致任何连接错误。大部分情况下，应当还需要下面内容：

DDRAW.LIB	DirectDraw 输入库函数
DINPUT.LIB	DirectInput 输入库函数
DSOUND.LIB	DirectSound 输入库函数

DMUSIC.LIB	DirectMusic 输入库函数
DSOUND3D.LIB	DirectSound3D 输入库函数
D3DIM.LIB	Direct3D 立即执行输入库函数
DXGUID.LIB	DirectX GUID 库函数
WINMM.LIB	Windows 多媒体扩展函数

对于上述文件，我们将在具体学习时再详细介绍，但是当你在连接过程中碰到“未知的符号 (unresolved symbol)”错误时应当记住使用这些库函数。我不希望收到初学程序员的关于这方面内容的电子邮件。

除了 DirectX .LIB 文件之外，还要在标题搜索路径中包含 DirectX .H 头文件，这一点同样要牢记在心。还要确认将 DirectX SDK 目录首先放在搜索路径列表中，因为许多 C++ 编译器含有老版本的 DirectX，并且在编译器本身的 INCLUDE\目录下也含有老版本的头文件，使用这些头文件是错误的。最合理的位置是 DirectX SDK include 目录下，该目录位于 INCLUDE\的 DirectX SDK 主安装目录下。

最后，如果使用 Borland 产品，要确认使用 DirectX .LIB 文件的 Borland 版本。该文件位于 DirectX SDK 安装程序中的 Borland\目录下。

目 录

序
简介

第一部分 Windows 编程基础

第一章 无尽之旅	3
历史一瞥	3
设计游戏	6
游戏类型	6
集思广益	7
设计文档和情节图板	8
使游戏具有趣味性	9
游戏的构成	9
常规游戏编程指导	14
使用工具	18
从准备到完成—使用编译器	20
实例: FreakOut	22
总结	39
第二章 Windows 编程模型	40
Windows 的历史	40
多任务和多线程	43
按照 Microsoft 方式编程: 匈牙利符号表示法	47
世界上最简单的 Windows 程序	50
真实的 Windows 应用程序	57
Windows 类	57
注册 Windows 类	64
创建窗口	64
事件处理程序	67
主事件循环	73
产生一个实时事件循环	78
打开多个窗口	79
总结	81

第三章 高级 Windows 编程	82
使用资源	82
使用菜单编程	101
图形设备接口 GDI 介绍	113
处理重要事件	127
将消息传递给自己	144
总结	146
第四章 Windows GDI、控件和突发奇想	147
高级 GDI 图形	147
点、线、平面多边形和圆	154
关于文本和字体	163
定时的重要性	164
使用控件	170
获取信息	177
T3D 游戏控制程序	183
总结	188

第二部分 DirectX 和 2D 基础

第五章 DirectX 基础和令人生畏的 COM	191
DirectX 基础	191
COM: 这是 Microsoft 的工作, 还是魔鬼的?	195
应用 DirectX COM 对象	207
COM 的前景	214
总结	214
第六章 首次接触: DirectDraw	215
DirectDraw 界面	215
创建 DirectDraw 对象	219
和 Windows 协同工作	223
进入事件模式	227
巧妙的色彩	230
创建一个显示画面	234
总结	255

第七章 高级 DirectDraw 和位图图形	256
真彩色模式下工作	256
双 缓 冲	269
动态画面	274
页面变换	277
应用图形变换器	283
剪切基础	295
采用位图	308
备用画面	319
位图的旋转和缩放	328
离散采样理论	330
色彩效果	334
人工色彩变换或者查询表.....	342
新的 DirectX 色彩和 Gamma 控制接口	343
GDI 和 DirectX 混合使用.....	344
获取 DirectDraw 的真相.....	346
在画面上冲浪	348
使用调色板	349
在窗口模式下应用 DirectDraw.....	349
总结	358
第八章 矢量光栅化及 2D 变换	359
绘制线条	359
线框多边形	384
2D 平面的变换.....	388
矩阵引论	401
变换	408
缩放	409
旋转	409
填充实心多边形	412
多边形碰撞检测	429
定时与同步详解	437
滚动和视角场景	439
伪 3D 等角引擎.....	445
T3DLIB1 库函数.....	449
BOB(变换对象)引擎.....	477
总结	485

第九章 用 DirectInput 和力反馈进行输入	486
输入循环回顾	486
DirectInput 序曲	488
力反馈详述	525
编写通用的输入系统: T3DLIB2.CPP	528
总结	534
第十章 用 DirectSound 和 DirectMusic 演奏乐曲	535
PC 上的声音编程	535
声音产生的原因	536
数字与 MIDI——发声大, 填充少	539
发声硬件	542
数字化记录: 工具和技术	543
DirectSound 中的麦克风	544
启动 DirectSound	546
主要与辅助的声音缓冲	549
播放声音	554
用 DirectSound 反馈信息	557
读取磁盘中数据	559
DirectMusic: 伟大的试验	563
DirectMusic 的结构	564
启动 DirectMusic	566
加载 MIDI 段	568
操作 MIDI 段	571
T3DLIB3 声音和音乐库	573
DirectSound API 封装	576
总结	585

第三部分 编程核心

第十一章 算法、数据结构、内存管理及多线程	589
数据结构	589
算法分析	598
递归	600
树结构	601
优化理论	611
制作演示程序	620

保存游戏的策略	621
实现多人游戏	623
多线程编程技术	624
总结	648
第十二章 人工智能在游戏中的运用.....	649
人工智能入门	649
明确的 AI 算法	650
模式和基础控制脚本	657
行为状态系统建模	663
应用软件对存储和学习建模.....	670
计划和决策树	672
导航	678
高级 AI 脚本	689
人工神经网络	695
遗传算法	698
模糊逻辑	700
在游戏中创建真正的 AI	718
小结	719
第十三章 基本物理建模	720
物理学基本定律	721
线性动量的物理性质：守恒和传递.....	731
万有引力效果模型	734
摩擦力	740
基本的特殊碰撞反应	746
2D 物体间的碰撞响应（高级）	757
解决 n-t 坐标系统	760
简单运动学	766
微粒系统	772
游戏关键：创建游戏的物理模型.....	781
总结	785
第十四章 综合运用	786
Outpost 的初步设计	786
编制游戏的工具	788
游戏领域：空间滚动	789

游戏者的飞船：“鬼怪号”	790
行星领域	792
敌人	794
供给能量	800
HUD	801
微粒系统	805
执行游戏	805
编译 Outpost.....	805
结束语	807

第四部分 附录

附录 A CD 上的内容	811
附录 B 安装 DirectX 和使用 C/C++ 编译器	813
使用 C/C++ 编译器.....	814
附录 C 三角函数和矢量	816
三角	816
矢量	819
附录 D C++ 基础	828
C++ 是什么	828
关于 C++ 应当了解的内容	831
新类型、关键词以及协议.....	831
内存管理	834
输入/输出流	834
类	836
作用域操作符	845
函数和操作符重载	846
小结	848
附录 E 游戏编程资源	849
游戏编程网站	849
下载站点	850
2D/3D 引擎.....	850
游戏编程书籍	850

Microsoft DirectX 多媒体展示	851
世界性新闻组网络系统	851
抓住产业：蓝调新闻	851
游戏开发杂志	852
游戏网站开发人员	852
Xtreme Games LLC	852
附录 F ASCII 表	854

第一部分

Windows 编程基础

第一章

无尽之旅

第二章

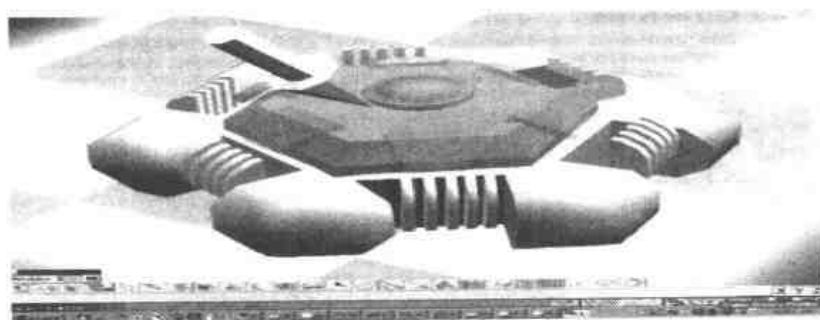
Windows 编程模型

第三章

高级 *Windows* 编程

第四章

WindowsGDI、控件和突发奇想



1

无尽之旅

Windows 编程是一场由来已久并还在进行着的战争。开始时，游戏程序拒绝 Windows 平台，但正如 Borg 所言：“反对无效……”，我也赞同这一观点。本章将就 Windows 的发展进行一下回顾。

- 游戏的历史
- 游戏类型
- 游戏编程的基本要素
- 使用工具
- 一个游戏的例子：FreakOut

历史一瞥

在 60 年代的某个时候，第一台台式计算机问世，当时运行在 Unix 机器上的 Core Wars，是最早的计算机游戏之一。当 70 年代的黄金岁月到来时，在全世界的台式计算机和小型计算机上流行着文本游戏和粗糙的图片游戏。有趣的是从那以后，大多数游戏开始网络化。我是说，90%的游戏程序是 MUD（Multi-User Dungeons）或类似的模拟游戏，如 Star Trek 和战争模拟游戏。但是，直到精彩的 Pong 游戏问世，大众才真正品尝到计算机游戏的乐趣。由 Nolan Busnell 设计的 Atari 计算机的问世，使单人的视觉游戏一夜之间充斥市场。

而后，在 1976~1978 年间，TRS-80、Apple、Atari 800 等型计算机开始冲击市场，这些是消费者买得起的第一代计算机。当然，读者也可以买类似 Altair 8000 型的组装机，但是又有谁乐意进行它们的组装呢？无论如何，这些计算机有各自的优缺点，Atari 800 是当

时功能最强大的计算机（我敢保证，我可以写一个 Wolfenstein 程序在它上面运行）。TRS-80 最商业化，而 Apple 机的市场最好。

慢慢地，游戏开始冲击计算机相关产业方面的市场，一夜之间，产生了许多年轻的百万富翁。制作者只需要一个类似“月亮领地”或 Pong 类型的好游戏，就可以突然致富！从那时，游戏开始像真正的计算机游戏，而且只有百十人知道如何编写游戏，当时绝对没有这类的指导书，只是有人不时半地下地出版一些 50~100 页的有许多令人费解之处的小册子，另外也曾在 Byte 杂志上有过文章，但是，大多时候，读者必须靠自己。

80 年代是游戏升温的年代。第一代 16 位计算机问世，如：IBM PC 及其兼容机、Mac、Atari ST、Amiga 500 等等。这是游戏变得好看的时候，甚至有些 3D 游戏如 Wing Commander、Flight Simulator 等开始出现在市场上。但是，PC 机肯定仍然落后于游戏机。在 1985 年以前，Amiga 500 和 Atair ST 作为最终游戏机占有绝对的支配地位。但是后来 PC 机由于低廉的价格和其商业用途开始大众化。并且最终必然将是具有最大市场基础的计算机——不考虑它的技术和质量——一统江湖。

在 90 年代初期，IBM PC 及其兼容机是主流。随着微软的 Windows 3.0 的发行，Apple Macintosh 寿终正寝。PC 是“工作者的计算机”。读者可以真正在上面进行游戏、编程、开机并连接其他东西。我想这就是为什么会有这么多人沉醉于 PC 而不是 Mac 的原因。一句话，读者不可能从 Mac 中寻求乐趣。

但是 PC 在图像或声音上还依然落后，PC 机看起来像没有足够的马力使游戏如同在 Amiga 或者游戏控制台上表现得那样好。

而后曙光终于来临……

在 1993 年后期，Id Software 发行了 DOOM 作为 Wolfenstein 3D（是由 Id 编写的第一批 3D 游戏之一）的换代产品。PC 机开始成为家用微机市场玩游戏或编程的选择，直到现在也是。DOOM 表明，只要读者有足够的才智，就可以在微机上面作任何事情。这点非常重要，记住，没有任何东西可以替代想像力和决心，只要读者认为可能，它就能实现。

在 DOOM 的疯狂冲击下，Microsoft 开始重新估计它在游戏和游戏编程上的地位。它意识到娱乐产业的巨大，并且只会更大。它开始制定巨大的计划涉足游戏，以使它得以在这个产业中分一杯羹。

问题是即使 Windows 95 的实时视听能力也表现得很恶劣，于是微软发行了 Win-G 以解决视觉方面的问题。Win-G 被宣布为游戏编程和图形子系统的最终解决方案，而实际上它不过只是用位图画的一些图像而已。一年之后 Microsoft 却又否认它的存在。——这可实有其事！

但是，新的图形、声音、输入、网络、3D 系统的工作已经开始，DirectX 诞生了。像以往一样，Microsoft 发行人员宣称它将解决世界上 PC 机平台上所有游戏编程的问题，并且在 Windows 上运行的游戏将同 DOS32 上的游戏一样快，甚至更快。但事实并非如此。

DirectX 开始的两个版本在实际应用中是失败的，但这并非指技术而言，Microsoft 低估了视觉游戏编程的复杂性及视觉游戏程序员能力。但是到了 DirectX3.0，DirectX 就开始工

作得比 DOS 出色！但是那时（1996~1997）许多公司仍然采用 DOS32 平台进行游戏编程，直到 DirectX 版本 5.0 发行，人们才转而使用 DirectX。

现在 DirectX 已经升级到 8.0 版本（本书包含 7.0 版本），它是一类极强大的 API。的确，读者应当改变一下想法——用 COM（Component Object Model）在 Win32 上编程，不再对整个计算机进行全部控制——但生活就是如此。

利用 DirectX 技术，读者可以创建一个 4GB 地址（或更多）、内存线性连续的、仿 DOS 的虚拟机，读者可以像在 DOS 环境（如果读者喜欢的话）下那样来编程。更为重要的是，现在读者可以即时操纵图像和声音技术的每一个新的片段。这都归功于 DirectX 远见卓识的设计和技术。总之，关于 DirectX 的话题已经足够多了，不久读者就会接触到整个处理所带来的效果。

首先出现的是 DOOM 游戏，它仅用到软件光栅技术，如图 1.1 所示。看一看 Rex Blade 游戏的屏幕表现，它是 DOOM 游戏的一个克隆版本。下一代的 3D 游戏，如 Quake I、Quake II 和 Unreal 就有了巨大的飞跃。再看一看图 1.2 中 Unreal 游戏的屏幕表现。这个游戏及其类似的游戏，其表现简直令人难以置信。所有这类游戏都采用软件光栅和硬件加速代码来获得最好的游戏表现。提请读者注意的是，Unreal 和 Quake II 游戏至少要运行在配置 Voodoo II 加速卡的 Pentium II 400MHz 机器上才能够达到这么眩目的效果。那么这将把我们带向何方？技术的发展越来越先进，以至空间已成为制约因素。然而，“奇迹”总会涌现。即使诸如 Quake 和 Unreal 这类的游戏需要花费数年才能制作完成，我仍然相信读者也能够创作出有如此吸引力的游戏。



图 1.1 Rex Blade: DOOM 技术第一代产品

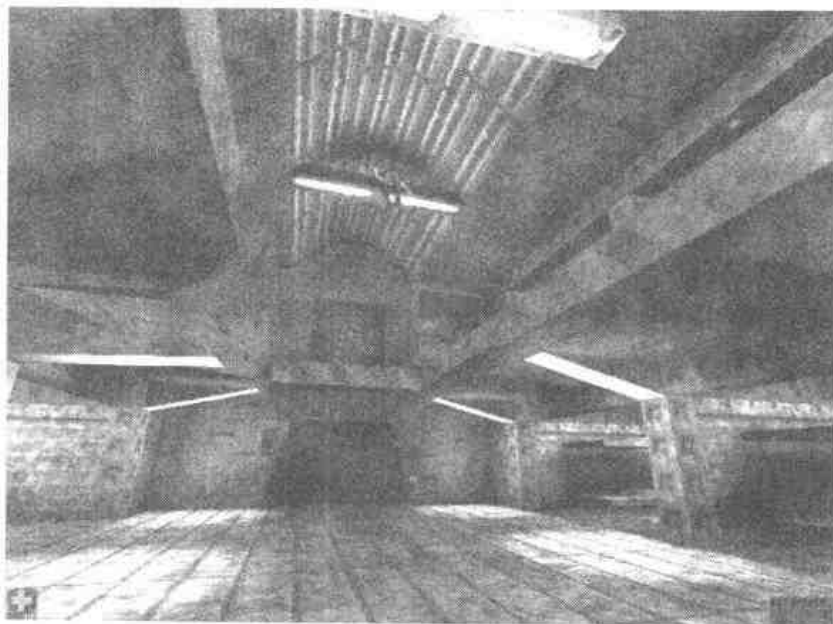


图 1.2 Unreal：效果奇佳

历史回顾到此为止，下面让我们转到游戏软件设计的核心上来。

设计游戏

编写视频游戏最难的一个工作是设计。的确，3D 运算很难，但是策划一个有趣的游戏并且进行设计可谓同样困难、重要。谁会关心游戏中是否采用最新的光子跟踪技术呢？

现在，构思一个游戏并不难。难的是它的细节体现、最终实现和相似物体在视觉形象上的差别表现。因此下面概述一下一些基本概念和本人在游戏设计工作中的一些心得、经验。

游戏类型

现在，游戏类型多如政治许诺（只说不做），但可以将它们归入以下几个类型：

类似 DOOM 的单人游戏——这些游戏大部分是全 3D 游戏，你将以角色的身份置身游戏中。DOOM、Hexen、Quake、Unreal、Duke Nukem 3D 以及 Dark Forces 都属于这一类型的游戏。

从技术上讲，它们或许是最难开发的游戏，这类游戏要求采用边缘切换技术。

运动游戏——运动游戏可以是 2D 的，也可以是 3D 的，但是近来 3D 的运动游戏越来越多了。运动游戏可以一个人玩，也可以多人玩。运动游戏的图像已经有了很大的改进。

也许运动游戏不像单人游戏给人印象深刻，但是它们也很吸引人。当然运动游戏中的人工智能水平却是所有游戏类别中最先进的。

格斗游戏——格斗游戏是典型的一人或两人玩的游戏。游戏动作采取在一旁观赏的角度或以一个移动的 3D 摄像机为观察角度。游戏人物一般为 2D、2.5D（3D 对象的多重 2D 位图映像）或全 3D 的。运行于索尼游戏平台上的 Tekken 是一款专为家庭消费平台市场而设计的游戏。格斗游戏不像 PC 机上的游戏那样流行，这或许是由于控制器界面问题以及需要二人同时参与所致。

街道/枪战/平面游戏——这些游戏是典型的 Asteroids、Pac-Man 和 Jazz Jackrabbit 类型素材，它们基本上都是老式游戏，并且主要是 2D 的，但是现在都被重新改造为 3D 游戏。而游戏规则和 2D 相同。

机械模拟游戏——这些游戏包括各种驾驶、飞行、赛艇、赛车、坦克战斗模拟以及读者能够想像到的任何其他种类，绝大部分都是 3D 游戏（直到最近游戏的表现才算差强人意）。

生态模拟游戏——这实际上是一种新游戏，除了现实世界本身外，没有其他类似物了。Populous、SimCity、SimAnt 等游戏均属此类。这类游戏让玩家作为一个主宰，控制某一种类的人工系统，可以是一个城市、一群蚂蚁、一个财政金融模拟，如 Gazzillionaire 游戏（一个很酷的游戏）。

攻略或战争游戏——这些游戏已经被分为许多小类。但是我对此并不赞成，而将其归为攻略和攻关（有时）更为恰当。Warcraft、Diablo、Final Fantasy VII 等皆属此类。这里我可能有点偏颇，因为 Diablo 虽是即时战略游戏，但它仍然含有大量的攻略和思考。另一方面，Final Fantasy 是攻关游戏而不是即时战略游戏。

交互式故事——这一类别包括 Myst-like 游戏。从根本上说，这些游戏都是预先“安排”或按照“路径”设计的，通过解决难题来过关。通常，因为缺少更好的定义，这些游戏不允许玩家自由地闲逛，就像玩人机交互的书一样。甚至，这些并不是真正的“to-the-metal”游戏程序，因为它们 99% 的都是用 Director 或类 Director 工具编写的，如 Boring、Jules。

重新流行的游戏——这类游戏一夜之间突然冒出来。简言之，人们又想玩一玩旧游戏，但游戏中情节和难度较原先复杂。例如，Atari 制作了大约 1000 个 Tempest 的版本。众所周知，它们从来就没有卖过，但是读者可以得到它们。非常荣幸，我对其中一些诸如 Dig Dug、Centipede、Frogger 等古老的游戏进行了重新制作。

纯智力和棋牌游戏——这里没有什么要说的东西，这些游戏可能是 2D、3D、预渲染或其他。Tetris、Monopoly 和 Mahjong 是这一类的游戏。

集思广益

一旦读者决定了制作哪一种游戏（这是件简单的事，因为我们知道自己喜欢什么），

就到了构思这个游戏的时候了。这完全由读者自己决定，策划一个好游戏并没有一成不变的思维定式。

首先，必须想出一个读者喜欢制作并将它开发成为听上去很酷的、可以实现的，并且其他人能够喜欢的游戏。

当然，读者可以将其他游戏作为模板或出发点来得到启发。不要简单地复制其他产品，但是大致模仿已成功的产品也是可以的。并且，大量阅读科幻书籍和游戏杂志，观察正在卖什么，观看大量的电影将有助于产生很酷的故事想法、游戏想法或视频动作。

我通常所做的是和朋友一起坐坐（或者是自己），抛出各种想法，直到出现听上去很酷的想法。然后，开发这个想法，直到似是而非，或者土崩瓦解为止。这令人非常心灰意冷。读者可能会对所有的想法厌烦，然后在两三个小时后放弃。不要灰心，这是件好事。如果一个游戏想法能存活一夜，第二天想起来还喜欢它的话，也许运气就来了。

警告



在这里我想给读者一个告诫：不要贪多嚼不烂！我已经收到上千封游戏编程新手的电子邮件。这些读者一心想在很短的时间内开发出诸如 DOOM 或 Quake 这类高水平的游戏来作为他们的处女作，显然这是不可能的。如果读者在 3~6 个月内能够完成一个 Asteroids 的复制品，就是很幸运的了，因此不要狂热。设定一个可行的目标，尝试考虑一些自己能做的事，因为最后可能只有自己在继续工作，别人都离你而去了。另外让处女作游戏的想法简单些。

下面我们进行一些细节工作。

设计文档和情节图板

一旦读者有了一个游戏想法，就应当将它落实到纸上。现在，当我进行一个大游戏产品时，需要自己编写一个实际的设计文档，但是对于少数游戏来讲，应当做几页的细节。首先一个设计文档是一个游戏的路标或框架。应当编写能想到的、尽可能多的游戏、游戏等级和游戏规则的细节。这样能知道正在做什么以及能按计划做下去。相反，如果想法一直在变化，那么所做的游戏将不连贯。

通常，我喜欢将一个简单的故事写下，开始的一或两页可能描写这个游戏是什么游戏、谁是主角、游戏思路是什么、最后游戏如何攻关。然后我决定游戏的核心细节——游戏等级和规则，列出尽可能多的细节。完成后，就一直进行添加或删除，但至少我有一个工作计划。如果想出了 100 条很酷的新思路，我能够一直补充它们，而不会忘记。

很明显，细节的数量由读者决定，但是要写下一一些东西，至少是一个游戏梗概。例如，可能读者不会连贯地考虑一个完整的设计文档，而是一些大致的游戏等级和规则的框图。图 1.3 是为一个游戏编写的情节图板的例子。没有复杂的细节，只有方便观察和工作的草图。

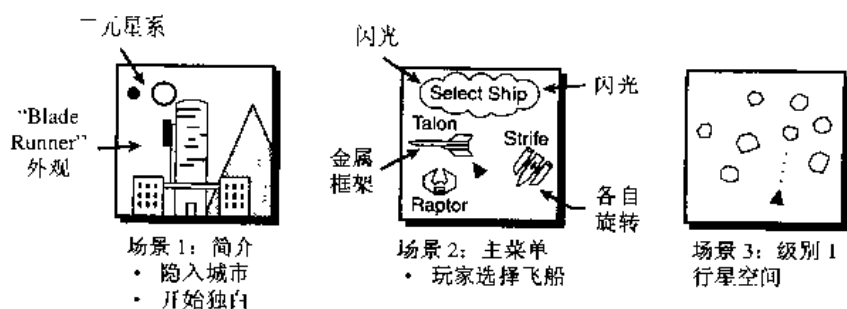


图 1.3 一个基本的情节串联图板

使游戏具有趣味性

游戏设计的最后部分是实际校验。读者确信游戏具有趣味性并且人们喜欢它吗？还是在自欺欺人？这是一个严重的问题。大约有 10000 个游戏被搁置，9900 个公司歇业，因此要仔细考虑。如果读者完全为之着迷并不顾一切的想玩这个游戏的话，那么已经大功告成。但是如果读者作为一个设计者对该想法冷淡的话，想像一下如何令其他人来对它感兴趣呢！

这里的关键是进行大量的游戏策划和 beta 测试，增加各种类型的非常酷的特性，因为这才是令游戏生动有趣的关键所在。这就如同橡木家具上的精美的手工艺品。人们喜欢这些游戏细节。

游戏的构成

现在来看一下一个视频游戏程序和其他各种程序的区别。视频游戏是一种极其复杂的软件，毫无疑问它们也是最难编写的程序。显然，编写 MS Word 程序要比 Asteroids 游戏难一点，但是编写 Unreal 游戏则要比我所能想像得到的其他任何程序都要难。

这就表示读者应当学习一种新的编程方式，这种方式更有益于实时应用和模拟，而不是读者经常使用的单行的、事件驱动的或顺序逻辑的程序。一个视频游戏基本上是一个连续的循环，它完成逻辑动作，并在屏幕上产生一个图像，通常是每秒钟 30 幅图或更多，这和电影的放映非常相似。只是读者要按自己的思路创建这个电影。

下面让我们从观察如图 1.4 所示的简化的游戏循环开始，下面对图中每个部分作些说明。

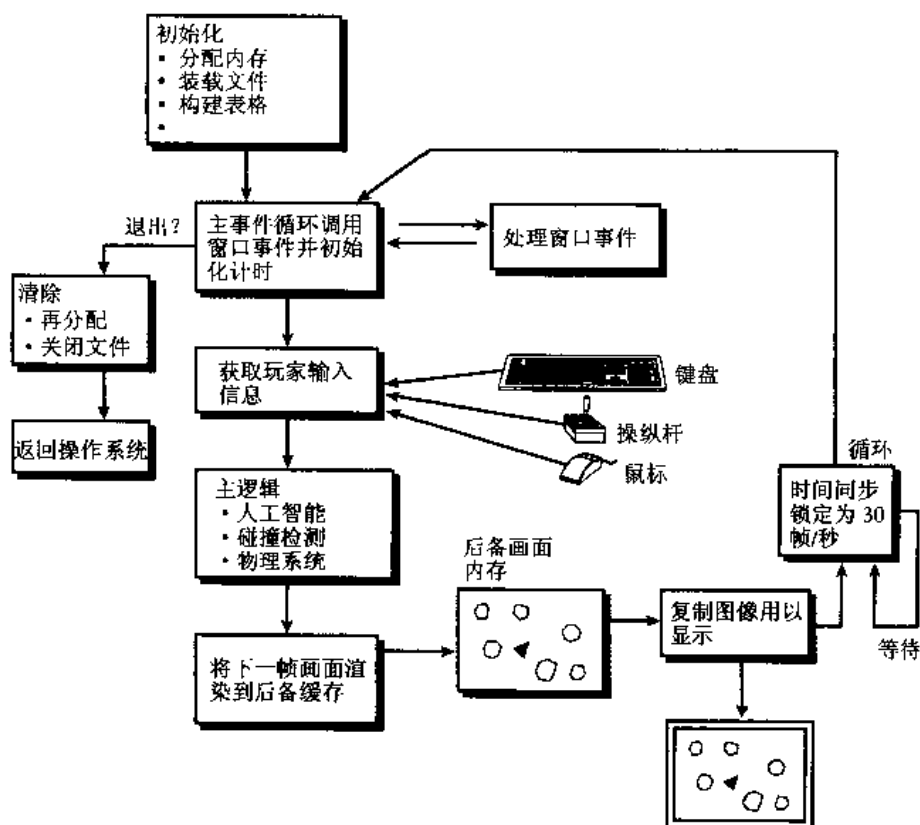


图 1.4 常规游戏循环结构

第一步：初始化

在这一步中，游戏程序运行的初始化操作和其他程序一样，如内存单元配置、资源采集、从磁盘装载数据等等。

第二步：进入游戏循环

在这一步中，代码运行进入游戏主循环，此时各种游戏动作和情节开始运行，直到用户退出游戏主循环。

第三步：获得玩家的输入信息

在这一步中，处理游戏玩家的输入信息并将其储存到缓存以备下一步人工智能和游戏逻辑使用。

第四步：执行人工智能和游戏逻辑

这部分包括游戏代码的主体部分，执行人工智能、物理系统和常规的游戏逻辑，其结果用于产生下一帧屏幕图像。

第五步：渲染下一帧图像

本步中，玩家输入和游戏人工智能和逻辑执行的结果，用来产生游戏的下一帧动画。这个图像通常放在后备缓存区内，因此无法看到它被渲染的过程。随后该图像被迅速拷贝到显示区中。

第六步：同步显示

许多计算机会因为游戏复杂程度的不同，游戏的速度会加快或减慢。例如，如果屏幕上有 1000 个对象在运行，CPU 的负载就比只有 10 个对象时重得多，因而游戏画面刷新速度也会有所改变，这是不允许的。因此必须确保游戏和最大帧速同步并使用定时器和/或等待函数来维持同步。一般认为 30 帧/秒是最佳的帧速。

第七步：循环

这一步非常简单，只需返回到游戏循环的入口并重新执行上述全部步骤。

第八步：关闭

这一步结束游戏，表示用户结束主体操作或游戏循环，返回操作系统。然而，在用户进行结束之前，用户必须释放所有的资源并刷新系统，这些操作和其他软件所进行的相应操作相同。

读者可能对上述游戏操作的细节感到疑惑。诚然，上面进行的解释有点过于简单化，但是它突出了如何进行游戏编程的重点。在大多数情况下，游戏循环是一个包括了大量状态的 FSM (Finite State Machine, 有限态计算机)。清单 1.1 是一个相当复杂的 C/C++ 游戏循环的实际代码。

程序清单 1.1 一个简单的游戏事件循环

```
// defines for game loop states
#define GAME_INIT           // the game is initializing
#define GAME_MENU          // the game is in the menu mode
#define GAME_STARTING      // the game is about to run
#define GAME_RUN           // the game is now running
```



```

#define GAME_RESTART    // the game is going to restart
#define GAME_EXIT      // the game is exiting

// game globals
int game_state = GAME_INIT; // start off in this state
int error      = 0;         // used to send errors back to OS

// main begins here

void main()
{
    // implementation of main game loop

    while (game_state!=GAME_EXIT)
    {
        // what state is game loop in
        switch(game_state)
        {
            case GAME_INIT: // the game is initializing
            {
                // allocate all memory and resources
                Init();

                // move to menu state
                game_state = GAME_MENU;
            } break;

            case GAME_MENU: // the game is in the menu mode
            {
                // call the main menu function and let it switch states
                game_state = Menu();

                // note: we could force a RUN state here
            } break;

            case GAME_STARTING: // the game is about to run
            {
                // this state is optional, but usually used to
                // set things up right before the game is run
                // you might do a little more housekeeping here
                Setup_For_Run();

                // switch to run state
                game_state = GAME_RUN;
            } break;

            case GAME_RUN: // the game is now running
            {

```

```
// this section contains the entire game logic loop

// clear the display
Clear();

// get the input
Get_Input();

// perform logic and ai
Do_Logic();

// display the next frame of animation
Render_Frame();

// synchronize the display
Wait();

// the only way that state can be changed is
// thru user interaction in the
// input section or by maybe losing the game.
} break;

case GAME_RESTART: // the game is restarting
{
    // this section is a cleanup state used to
    // fix up any loose ends before
    // running again
    Fixup();
    // switch states back to the menu
    game_state = GAME_MENU;
} break;

case GAME_EXIT: // the game is exiting
{
    // if the game is in this state then
    // it's time to bail, kill everything
    // and cross your fingers
    Release_And_Cleanup();

    // set the error word to whatever
    error = 0;

    // note: we don't have to switch states
    // since we are already in this state
    // on the next loop iteration the code
    // will fall out of the main while and
    // exit back to the OS
} break;
```

```
        default: break;  
    } // end switch  
  
    } // end while  
  
    // return error code to operating system  
    return(error);  
  
} // end main
```

尽管清单 1.1 是一个没有任何功能的程序，但通过学习其游戏循环的结构可以获得好的思路。所有的游戏循环大都按照这个结构的一种形式或另一种形式进行设计。图 1.5 表示了游戏循环逻辑的状态转换图。如读者所见，状态转换是非常连贯的。

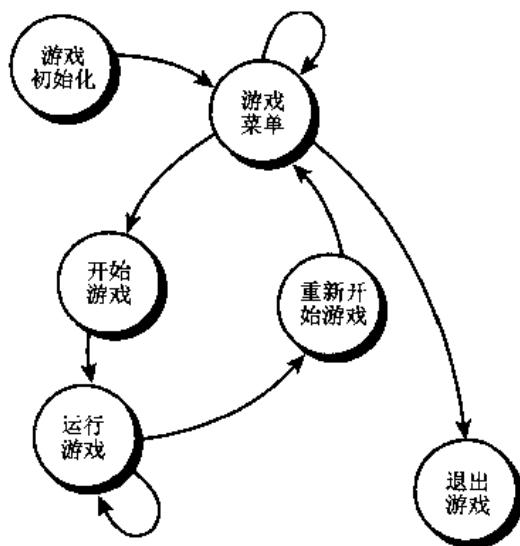


图 1.5 一个游戏循环的状态转换图

关于游戏循环和有限态计算机（FSM）的内容将在本章最后涉及 FreakOut 演示游戏的章节中再进行更详细的讨论。

常规游戏编程指导

下面讨论一下读者所关心、也是游戏编程常用的技术和基本原理，这有利于简化游戏编程的复杂程度。

首先，视频游戏是运行于超高性能计算机上的游戏程序。对于时间或内存要求特别严格的代码部分不能使用高级 API 来编程，和游戏代码内部循环有关的部分，大都需自己手

工编写, 否则游戏将会碰到严重的速度和性能问题。当然, 这并不意味着就不能信任 DirectX 等 API 编程工具, 因为 DirectX 以高性能和尽可能“瘦”的方式来编写。但在通常情况下, 要避免高级的函数调用。

除上述情况应多加注意外, 在编程时还应留意下面的编程技巧。

技巧



不要怕使用全局变量。许多视频游戏不使用大量的带有形参的、与时间相关的函数, 而是使用一个全局变量来代替。例如一个函数的代码如下:

```
void Plot (int x, int y, int color)
{
    //在屏幕上画一个点像素
    video_buffer[x + y*MEMORY_PITCH] = color;
} // 结束 Plot
```

函数体运行的时间小于函数调用所需的时间, 这是由于参数压入和弹出堆栈造成的。在这种情况下, 更好的方法可能是创建一个全局变量, 然后在调用前进行赋值, 像下面一样:

```
int gx,gy,gz,gcolor; //定义一些全局变量

void Plot_G(void)
{
    //使用全局变量来画一个点像素
    video_buffer[gx + gy*MEMORY_PITCH] = gcolor;

} //结束 Plot_G
```

技巧



使用内联功能。通过使用内联指令来完全摆脱调用功能甚至能够改善上面的技巧。内联指令不调用函数, 而指示编译器将被调用函数代码放在需要调用该函数的最佳位置, 这样做会使程序变得更大, 但却提高了运行速度。下面是一个实例:

```
inline void Plot_I (int x, int y, int color)
{
    //在屏幕上画一个点像素
    video_buffer[x + y*MEMORY_PITCH] = color;
} // 结束 Plot_I
```

注意: 这里并没有使用全局变量, 因为编译器有效运行了相同类型数据的别名。但是全局变量迟早会派上用场。如果在函数调用时, 一个或两个形参已改变, 由于没有重新加载, 所以旧的参数值有可能仍被使用。

技巧



尽量使用 32 位变量而不用 8 位变量或 16 变量。Pentium 和之后的处理器全部都是 32 位的, 这就意味着它们并不喜欢 8 位或 16 位的数据字符, 实际上, 更小的数据可能会由于超高速缓存和其他相关的内存寻址异常而使速度下降。例如, 读者可能创建一个如下所示的结构:

```
struct CPOINT
{
    short x,y;
    unsigned char c;
} // 结束 CPOINT
```

技巧



尽管这个结构看上去很好，但实际并非如此！首先，结构本身目前是一个 5 字节长的结构——2 个 short+1 个 char=5。这实际上是一个很差的设计，这将导致内存地址崩溃。更好的结构形式如下：

```
struct CPOINT
{
    int x,y;
    int c;
} // 结束 CPOINT
```

C++

提示：除了默认的公共可见性外，C++ 中的结构更像是“类”。

这种新结构要更好一些。首先，所有的成员都是相同尺寸——也就是说整数的大小为 4 字节。因此，单个指针可以通过递增 **DWORD**（双字节）的边界访问任何单元。当然，这种新结构的大小就是 3 个整数长，即 12 字节，至少是 4 的倍数，或者在 **DWORD** 边界上。这样将明显地提升性能。

实际上，如果读者真想稳妥的话，应当将所有的结构都变为 32 字节的倍数。由于 Pentium 家族处理器芯片上标准缓存线长度是 32 的倍数，因而这是一个最佳长度。可以通过人工虚设单元或者使用编辑指令（最简单的方法）来满足这个要求。当然，这可能会浪费大量的内存，但是为了提高速度这是值得的。

技巧



注释你的代码。游戏程序员不注释代码是出了名的。不要犯相同的错误。用额外的输入换取整洁。注释良好的代码是值得的。

技巧



程序应以类似 RISC（精简指令系统计算机）的形式来编写。换句话说，尽量简化你的代码，而不是使它更复杂。Pentium 和 Pentium II 处理器特别喜欢简单指令，而不是复杂的指令。你的程序可以长些，但应尽量使用简单指令，使程序相对于编辑器来说更加简单些。例如，不要编写类似下面的程序：

```
if ((x+=(2*buffer[index++]))>10)
{
    //进行工作
} // 结束 if
```

应这样做：

```
x+=(2*buffer[index]);
index++;

if (x > 10)
{
    //进行工作
} // 结束 if
```

技巧



按照这种方式来编写代码有两个原因。首先，它允许调试程序在代码各部分之间设置断点；第二，这将更易于编译器向 Pentium 处理器传送简单指令，这样将使处理器使用更多的执行单元并行地处理更多的代码。复杂的代码就比较糟糕。

技巧



对于简单的、是 2 的倍数的整数乘法运算，应使用二进制移位运算。因为所有的数在计算机中都以二进制存储，位组合向左或右移动就等同于乘法和除法运算。例如：

```
int y_pos = 10

//将 y_pos 乘以 64
y_pos = (y_pos << 6); //2^6=64
```

相似的有：

```
{
//将 y_pos 除以 8
y_pos = (y_pos >> 3); //1/2^3=1/8
```

当读者接触到优化那一章时，将会发现更多的、类似的技巧。哈哈，酷吧！

技巧



编写高效的算法，世界上所有的汇编语言都不会将 $n+2$ 算法运行得更快些，更好的方法是使用整齐、高效的算法而不是蛮干。

技巧



不要在编程过程中优化代码。这通常会浪费时间，等到完成主要的代码块或整个程序后才开始进行繁重的优化工作。这样可以节省你的时间，因为你必须处理一些模糊的代码或不必要的优化。当游戏编程完成时，才到了剖析代码、查找问题以优化程序的时间。另一方面，程序要注意错落有致，不要杂乱无章。

技巧



不要为简单的对象编写大量的复杂的数据结构。仅仅因为连接的清单非常酷并不意味着必须使用它们。对于静态数组而言，其元素一般为 256 个，所以只需为之静态分配内存并进行相应的处理即可。视频游戏编程 90% 都是数据操作。游戏程序的数据应尽可能简单、可见，以便能够迅速地存取它，随意操作它或进行其他处理。确保你的数据结构按照这一原则进行处理。

技巧



使用 C++ 应谨慎。如果读者是位老练的高手，继续前进去做你喜欢的事，但是不要去疯狂追求爽，或使游戏程序过于复杂以至于超出一般计算机的承受能力。简单、直观的代码是最好的程序，也最容易调试。我从来都不想有多重的隶属关系。

技 巧



如果感到前路荆棘丛生，那就停下来，回头然后绕路而行。我见过许多游戏程序员开始于一条很差的编程路线，然后葬送自己。能够意识到自己所犯的错误，然后重新编写 500 行的代码要比得到一个不是期望的代码结构要好得多。因此，如果在工作中发现问题，那就要重新评估并确保它是值得花时间补救的。

技 巧



经常备份你的工作。在编写游戏代码时，需要相当频繁地锁定系统。重新做一个排序算法比较容易，但是要为一个新角色或碰撞检测重新编写 AI 则是另一回事。

技 巧



在开始你的游戏项目之前，应当进行一下组织工作。使用合理的文件名和目录名，提出一种一致的变量命名约定，尽量对图形和声音数据使用分开的目录，而不是将其全部都放置在一个目录中。

使用工具

过去编写视频游戏通常只不过需要一个文本编辑器和一个简略的自制图形程序。但是现在事情就变得复杂一点了，读者至少需要一个 C/C++ 编译器、一个 2D 的图形程序和一个声音处理程序。此外，如果读者想编写一个 3D 游戏的话，读者可能还需要一个 3D 的模型，而如果读者想使用任何 MIDI 设备的话，还要有一个音乐排序程序。

让我们来浏览一下目前流行的产品及其功用。

C/C++ 编译器

对于 Windows 9X/NT 的研制来讲，简直没有比 MS VC++5.0+ 更好的编译器了。它可以做任何读者想做的事，甚至更多。所产生的 .EXE 文件是最快的有效代码。Borland 编译器也可以工作得很好（并且它要便宜得多），但是它的特性设置较少。在任何情况下，读者不一定需要上述任何一种编译器的增强版本，一个能够产生 Win32 平台下的 .EXE 文件的学生版本就已经足够了。

2D 艺术软件

这儿读者可以得到图形程序、画图程序和图像处理程序。图形程序主要允许读者以原色一个像素、一个像素地绘制和变换图形。直到现在为止，JASC 公司的 Paint Shop Pro5.0+ 还是性价比最佳的软件包。Fractal Design Painter 也很好，但是它更适用于传统的艺术家，而且它很昂贵。我喜欢使用 Corel Photo-Paint，但是对于网络游戏新手的需要来讲，它的功能的确有点偏大。

另一方面，画图程序允许读者创建图像，通常用曲线、直线和 2D 的几何原型来创建图像。这些程序并不是很有用，但如果读者需要的话，Adobe Illustrator 是一个很好的选择。

2D 艺术程序中的最后一类是图像处理类。这些程序多用于产品的后期制作，而不是前期的艺术创建。Adobe Photoshop 是大多数人喜欢的软件，但是我认为 Corel Photo-Paint 更好一些。读者自己来决定吧。

声音处理软件

目前用于游戏的所有的声音效果 (SFX) 90% 都是数字样本，采用这种类型的声音数据来工作，读者应当需要一个数字声音处理程序。这一类中最好的程序是 Sound Forge Xp。目前为止，它具有我所见到的最复杂的声音处理能力，并且使用也最方便。

3D 造型软件

这是挑战经济实力的软件。3D 造型软件价格可能需要上万美金，但是最近也有大量的低价的 3D 造型软件上市，并且也具有足够的功能来制作一部影片。我主要是使用简单的中等复杂程度的 3D 造型和动画软件——Caligari trueSpace III+。在这种价位上，这是最好的 3D 造型软件，只要几百美金，并且拥有最好的界面。

如果读者希望功能更加强大并追求绝对的超级现实主义，3D Studio Max II+ 就可以做到这一点，但是它的价格大约要 2500 美金，因此应当认真考虑一下。然而如果我们使用这些造型软件只是用来创建 3D 的网格，而不是渲染，那么所有的其他修饰也就不需要了。这样 trueSpace 就足以应付。

音乐和 MIDI 排序程序

目前的游戏中有两类音乐：纯数字式（像 CD 一样）和 MIDI（乐器数字界面）式，MIDI 是一种基于人工记录数据的合成音效。如果想制作 MIDI 信息和歌曲，读者还需要一个排序软件包。一种性价比最佳的软件包是 Cakewalk，因此，如果读者打算记录和制作 MIDI 音乐的话，建议最好去了解一下这个软件。在涉及 DirectMusic 内容的第 10 章“用 DirectSound 和 DirectMusic 演奏乐曲”中，我们将对 MIDI 数据再作探讨。

技巧



现在是最酷的部分……许多软件制造商将许我在 CD 上列出了它们软件的共享版或评测版，赶快去体验这些软件吧！

从准备到完成——使用编译器

学习 Windows 游戏编程的一件最容易令人灰心丧气的事是学习如何使用编译器。大多数情况下，读者对开始编写游戏程序如此激动，以至于全身心地投入到 IDE（集成开发环境）中去并尝试进行编译，然后就出现了一百万条编译和连接错误！为了有助于解决这个问题，让我们首先回顾一下有关编译器的一些基本概念。

0. 请读者务必阅读全部的编译器指令！
1. 必须在你的系统上安装 DirectX SDK（软件开发工具包）。你所要做的就是到 CD 上查找到 <DirectX SDK> 目录，阅读 README.TXT 文件，并按说明进行操作（实际上只不过是“单击 DirectX SDK INSTALL.EXE 程序”）。
2. 我们要制作的是 Win32 .EXE 程序，而不是 .DLL 文件和 ActiveX 组件等等。因此如果读者想编译的话，需要做的第一件事情是使用编译器创建一个新的工程或工作区，然后将目标输出文件设定为 Win32 .EXE。使用 VC++5.0 编译器进行这一步的工作如图 1.6 所示。

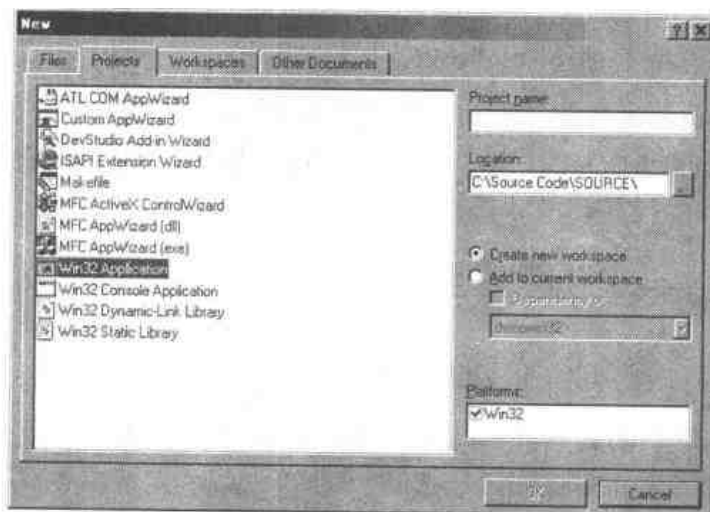


图 1.6 使用 Visual C++5.0 创建一个 Win32 .EXE 文件

3. 应用添加文件（ADD Files）命令从主菜单或工程节点本身向工程添加源文件。对于使用 VC++5.0 编译器而言，其操作过程如图 1.7 所示。
4. 从接触到 DirectX 一章时起，就必须包含下面列出的和图 1.8 所表示的 DirectX COM 界面库文件。
 - DDRAW.LIB
 - DSOUND.LIB
 - DSOUND3D.LIB

- DINPUT.LIB
- DMUSIC.LIB
- DSETUP.LIB



图 1.7 利用 VC++5.0 向一个工程添加文件

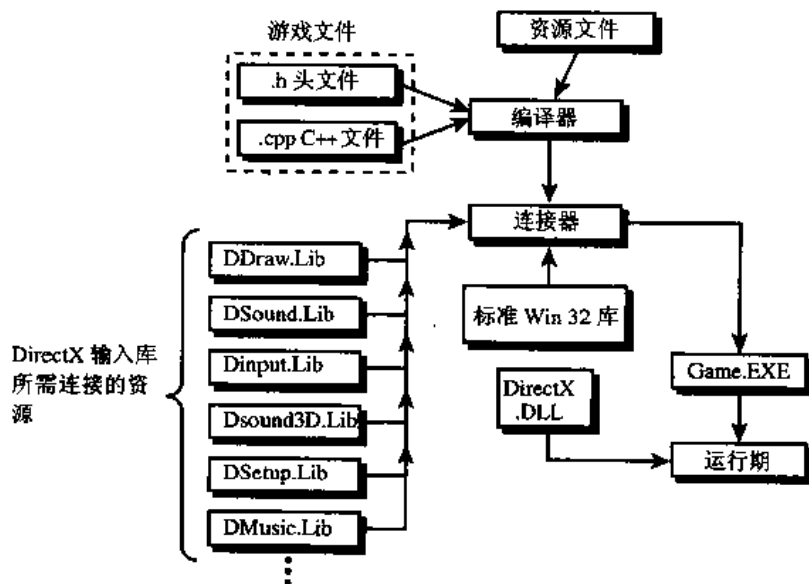


图 1.8 创建 Win32 .EXE 应用程序所需要的资源

这些 DirectX .LIB 文件位于所安装的 DirectX SDK 根目录下的 <LIB> 目录下。必须将这些 .LIB 库文件添加到读者的工程或工作区中。读者不可能只添加搜索路径，因为搜索引擎会发现编译器本身安装的库文件和旧的 DirectX 3.0 的 .LIB 文件。如果是这样做的话，读者可能不得不将 Windows 多媒体扩展库文件——WINMM.LIB 加入到工程中去。该文件位于读者的编译器安装目录下的 <LIB> 目录下。

5. 准备编译你的程序。

警告



如果读者是 Borland 用户，在 DirectX 软件开发工具包中有一个单独的 Borland 库文件目录。因此要确保将这些 .LIB 文件而不是目录树中上一级的 MS 兼容文件添加到工程中。

如果读者仍然对此有疑问的话，请不必着急。在本书中，当讨论 Windows 编程和首次接触 DirectX 时还要多次重复这些步骤。

实例：FreakOut

在沉溺于所讨论的有关 Windows、DirectX 和 3D 图形之前，应当暂停一下，先显示一个完整的游戏——虽然简单了一点，但毫无疑问是一个完整的游戏。读者可以看到一个真实的游戏循环和一些图形的调用以及一些编码工作。怎么样？跟我来吧！

问题是我们现在仅仅在第一章。我并不喜欢使用后面章节中的内容……这听起来像在欺骗读者，对吧？因此，我决定要做的是使用以前常常使用的黑匣子 API 来进行游戏编程。基于这个要求，我要提一个问题“创建一个类似 Freakout 的 2D 游戏，其最低要求是什么？”我们真正所需要的是下面的功能：

- 转换为任何图像模式
- 在屏幕上画各种颜色的矩形
- 采用键盘输入
- 使用定时函数同步游戏循环
- 在屏幕上画一串带颜色的文本

因此我创建一个名字为 BLACKBOX.CPPH 的目录。该程序带有一套 DirectX 函数集，并且包含实现所需要功能的支持代码。最妙的是，读者根本不需要去查看这些代码，只要在这些函数原型的基础上使用这些函数就可以了，并确保连接上 BLACKBOX.CPPH 文件来产生一个 .EXE 文件。

以 BLACKBOX 库为基础，我编写了一个名字为 FreakOut 的游戏，这个游戏演示了本章中所讨论的许多概念。FreakOut 游戏含有实际游戏的全部主要组成部分，包括：一个游戏循环、计分、等级，甚至还带有一个球的小型物理模型。我所说的是小型的模型。图 1.9 是一幅游戏运行中的屏幕画面。当然它远远不及 Arkanoid，但 4 个小时的工作有此成果也

不赖！

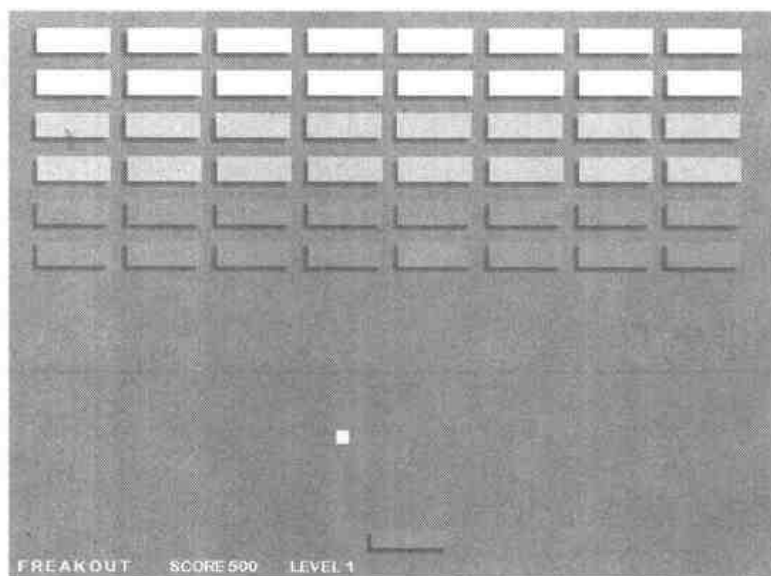


图 1.9 FreakOut 的屏幕攻关

在编写游戏源代码之前，我希望读者能看一下工程和其各种构件是如何协调一致的。参见图 1.10。

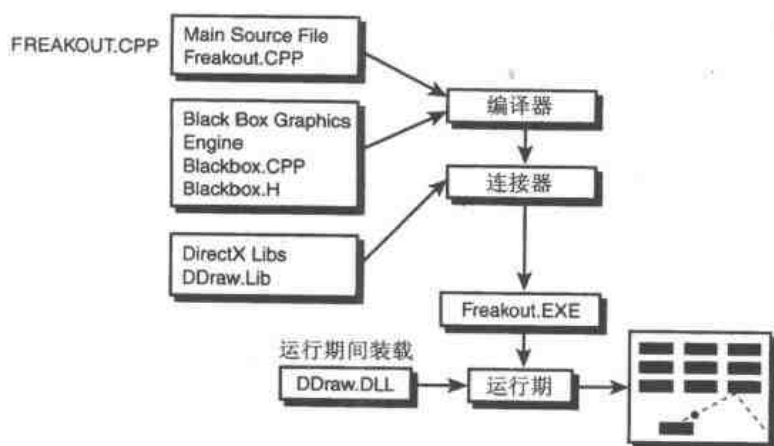


图 1.10 FreakOut 的结构

从图中可以看到，游戏主要由下面文件构成：

FREAKOUT.CPP——主要的游戏逻辑，使用 **BLACKBOX.CPP**，创建一个最小的 Win32 应用程序。

BLACKBOX.CPP——游戏库（请不要偷看）。

BLACKBOX.H——游戏库的头文件。

DDRAW.LIB——用来建立应用程序的 DirectDraw 输入库。其中并不含有真正的 DirectX

代码。主要用来作为一个函数调用的中间库，可以轮流调用进行实际工作的 DDRAW.DLL 动态连接库。它可以在 DirectX SDK 安装程序的<LIB>目录下找到。

DDRAW.DLL——运行过程中的 DirectDraw 库，实际上含有通过 DDRAW.LIB 输入库调用 DirectDraw 界面函数的 COM 执行程序。对此读者不必担心；只要确认已经安装了 DirectX 运行文件即可。

现在，我们对此已有了了解，让我们看一下 BLACKOUT.H 头文件，看看它包含了哪些函数。

程序清单 1.2 BLACKOUT.H 头文件

```
// BLACKBOX.H - Header file for demo game engine library

// watch for multiple inclusions
#ifndef BLACKBOX
#define BLACKBOX

// DEFINES ////////////////////////////////////////

// default screen size

#define SCREEN_WIDTH    640 // size of screen
#define SCREEN_HEIGHT   480
#define SCREEN_BPP      8   // bits per pixel
#define MAX_COLORS      256 // maximum colors

// MACROS ////////////////////////////////////////

// these read the keyboard asynchronously
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code)   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// initializes a direct draw struct
#define DD_INIT_STRUCT(ddstruct) { memset(&ddstruct,0,sizeof(ddstruct));\
ddstruct.dwSize=sizeof(ddstruct); }

// TYPES ////////////////////////////////////////

// basic unsigned types
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char  UCHAR;
typedef unsigned char  BYTE;

// EXTERNALS ////////////////////////////////////////

extern LPDIRECTDRAW4      lpdd;          // dd object
extern LPDIRECTDRAWSURFACE4 lpddsprimary; // dd primary surface
```

```

extern LPDIRECTDRAW_SURFACE4 lpddsback;    // dd back surface
extern LPDIRECTDRAW_PALETTE lpddpal;      // a pointer to palette
extern LPDIRECTDRAW_CLIPPER lpddclipper;  // dd clipper
extern PALETTEENTRY palette[256];        // color palette
extern PALETTEENTRY save_palette[256];    // used to save palettes
extern DDSURFACEDESC2 ddsd;              // dd surface description struct
extern DDBLTFX ddbltfx;                  // used to fill
extern DDSCAPS2 ddscaps;                 // dd surface capabilities struct
extern HRESULT ddrval;                   // result back from dd calls
extern DWORD start_clock_count;          // used for timing

// these defined the general clipping rectangle
extern int min_clip_x, // clipping rectangle
          max_clip_x,
          min_clip_y,
          max_clip_y;

// these are overwritten globally by DD_Init()
extern int screen_width, // width of screen
          screen_height, // height of screen
          screen_bpp;    // bits per pixel

// PROTOTYPES //////////////////////////////////////

// DirectDraw functions
int DD_Init(int width, int height, int bpp);
int DD_Shutdown(void);
LPDIRECTDRAW_CLIPPER DD_Attach_Clipper(LPDIRECTDRAW_SURFACE4 lpdds,
                                       int num_rects, LPRECT clip_list);
int DD_Flip(void);
int DD_Fill_Surface(LPDIRECTDRAW_SURFACE4 lpdds, int color);

// general utility functions
DWORD Start_Clock(void);
DWORD Get_Clock(void);
DWORD Wait_Clock(DWORD count);

// graphics functions
int Draw_Rectangle(int x1, int y1,
                  int x2, int y2,
                  int color,
                  LPDIRECTDRAW_SURFACE4 lpdds=lpddsback);

// gdi functions
int Draw_Text_GDI(char *text, int x, int y, COLORREF color,
                  LPDIRECTDRAW_SURFACE4 lpdds=lpddsback);
int Draw_Text_GDI(char *text, int x, int y, int color,
                  LPDIRECTDRAW_SURFACE4 lpdds=lpddsback);

#endif

```

现在，就不要将你的精力都浪费在程序代码和那些神秘的全局变量是什么的问题上。让我们来看一看这些函数。如读者所看到一样，这个简单的图形界面需要一些函数来完成。在这个图形和小型的 Win32 应用程序（我们要做的 Windows 编程工作越少越好）的基础上，我创建了游戏 FREAKOUT.CPP，如实例 1.3 所示。请认真地看一看，特别是主游戏循环和对游戏处理功能的访问。

程序清单 1.3 FREAKOUT.CPP 源文件

```
// INCLUDES ////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // include all macros
#define INITGUID           // include all GUIDs

#include <windows.h>        // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>

#include <iostream.h>       // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h>          // directX includes
#include "blackbox.h" // game library includes

// DEFINES ////////////////////////////////////////

// defines for windows
#define WINDOW_CLASS_NAME "WIN3DCLASS" // class name

#define WINDOW_WIDTH      640    // size of window
#define WINDOW_HEIGHT     480

// states for game loop
#define GAME_STATE_INIT    0
#define GAME_STATE_START_LEVEL 1
#define GAME_STATE_RUN     2
#define GAME_STATE_SHUTDOWN 3
#define GAME_STATE_EXIT    4
```

```

// block defines
#define NUM_BLOCK_ROWS      6
#define NUM_BLOCK_COLUMNS   8

#define BLOCK_WIDTH          64
#define BLOCK_HEIGHT        16
#define BLOCK_ORIGIN_X      8
#define BLOCK_ORIGIN_Y      8
#define BLOCK_X_GAP         80
#define BLOCK_Y_GAP         32

// paddle defines
#define PADDLE_START_X      (SCREEN_WIDTH/2 - 16)
#define PADDLE_START_Y      (SCREEN_HEIGHT - 32);
#define PADDLE_WIDTH        32
#define PADDLE_HEIGHT       8
#define PADDLE_COLOR        191

// ball defines
#define BALL_START_Y        (SCREEN_HEIGHT/2)
#define BALL_SIZE           4

// PROTOTYPES //////////////////////////////////////

// game console
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// GLOBALS //////////////////////////////////////

HWND main_window_handle = NULL; // save the window handle
HINSTANCE main_instance = NULL; // save the instance
int game_state          = GAME_STATE_INIT; // starting state

int paddle_x = 0, paddle_y = 0; // tracks position of paddle
int ball_x   = 0, ball_y   = 0; // tracks position of ball
int ball_dx  = 0, ball_dy  = 0; // velocity of ball
int score    = 0;           // the score
int level    = 1;           // the current level
int blocks_hit = 0;         // tracks number of blocks hit

// this contains the game grid data

UCHAR blocks[NUM_BLOCK_ROWS][NUM_BLOCK_COLUMNS];

// FUNCTIONS //////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,

```



```

        UINT msg,
        WPARAM wparam,
        LPARAM lparam)
{
    // this is the main message handler of the system
    PAINTSTRUCT ps;           // used in WM_PAINT
    HDC          hdc;         // handle to a device context

    // what is the message
    switch(msg)
    {
        case WM_CREATE:
        {
            // do initialization stuff here
            return(0);
        } break;

        case WM_PAINT:
        {
            // start painting
            hdc = BeginPaint(hwnd,&ps);

            // the window is now validated

            // end painting
            EndPaint(hwnd,&ps);
            return(0);
        } break;

        case WM_DESTROY:
        {
            // kill the application
            PostQuitMessage(0);
            return(0);
        } break;

        default:break;

    } // end switch

    // process any messages that we didn't take care of
    return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////

int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,

```

```

        LPSTR lp cmdline,
        int ncmdshow)
{
    // this is the winmain function

    WNDCLASS winclass; // this will hold the class we create
    HWND      hwnd;    // generic window handle
    MSG        msg;     // generic message
    HDC        hdc;     // generic dc
    PAINTSTRUCT ps;     // generic paintstruct

    // first fill in the window class structure
    winclass.style = CS_DBLCLKS | CS_OWNDC |
                    CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra = 0;
    winclass.cbWndExtra = 0;
    winclass.hInstance = hinstance;
    winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;

    // register the window class
    if (!RegisterClass(&winclass))
        return(0);

    // create the window, note the use of WS_POPUP
    if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
        "WIN3D Game Console", // title
        WS_POPUP | WS_VISIBLE,
        0,0, // initial x,y
        GetSystemMetrics(SM_CXSCREEN), // initial width
        GetSystemMetrics(SM_CYSCREEN), // initial height
        NULL, // handle to parent
        NULL, // handle to menu
        hinstance, // instance
        NULL))) // creation parms
        return(0);

    // hide mouse
    ShowCursor(FALSE);

    // save the window handle and instance in a global
    main_window_handle = hwnd;
    main_instance = hinstance;

    // perform all game console specific initialization

```

```

Game_Init();

// enter main event loop
while(1)
{
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
        // test if this is a quit
        if (msg.message == WM_QUIT)
            break;

        // translate any accelerator keys
        TranslateMessage(&msg);

        // send the message to the window proc
        DispatchMessage(&msg);
    } // end if

    // main game processing goes here
    Game_Main();

} // end while

// shutdown game and release all resources
Game_Shutdown();

// show mouse
ShowCursor(TRUE);

// return to Windows like this
return(msg.wParam);

} // end WinMain

// T3DX GAME PROGRAMMING CONSOLE FUNCTIONS //////////////////////////////////

int Game_Init(void *parms)
{
    // this function is where you do all the initialization
    // for your game

    // return success
    return(1);

} // end Game_Init

////////////////////////////////////

int Game_Shutdown(void *parms)

```

```

{
// this function is where you shutdown your game and
// release all resources that you allocated

// return success
return(1);

} // end Game_Shutdown

////////////////////////////////////

void Init_Blocks(void)
{
// initialize the block field
for (int row=0; row < NUM_BLOCK_ROWS; row++)
    for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
        blocks[row][col] = row*16+col*3+16;

} // end Init_Blocks

////////////////////////////////////

void Draw_Blocks(void)
{
// this function draws all the blocks in row major form
int x1 = BLOCK_ORIGIN_X, // used to track current position
    y1 = BLOCK_ORIGIN_Y;

// draw all the blocks
for (int row=0; row < NUM_BLOCK_ROWS; row++)
{
    // reset column position
    x1 = BLOCK_ORIGIN_X;

    // draw this row of blocks
    for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
    {
        // draw next block (if there is one)
        if (blocks[row][col]!=0)
        {
            // draw block
            Draw_Rectangle(x1-4,y1+4,
                           x1+BLOCK_WIDTH-4,y1+BLOCK_HEIGHT+4,0);

            Draw_Rectangle(x1,y1,x1+BLOCK_WIDTH,
                           y1+BLOCK_HEIGHT,blocks[row][col]);
        } // end if

        // advance column position
    }
}

```

```

        x1+=BLOCK_X_GAP;
    } // end for col

    // advance to next row position
    y1+=BLOCK_Y_GAP;

    } // end for row

} // end Draw_Blocks

////////////////////////////////////

void Process_Ball(void)
{
    // this function tests if the ball has hit a block or the paddle
    // if so, the ball is bounced and the block is removed from
    // the playfield note: very cheesy collision algorithm :)

    // first test for ball block collisions

    // the algorithm basically tests the ball against each
    // block's bounding box this is inefficient, but easy to
    // implement, later we'll see a better way

    int x1 = BLOCK_ORIGIN_X, // current rendering position
        y1 = BLOCK_ORIGIN_Y;

    int ball_cx = ball_x+(BALL_SIZE/2), // computer center of ball
        ball_cy = ball_y+(BALL_SIZE/2);

    // test of the ball has hit the paddle
    if (ball_y > (SCREEN_HEIGHT/2) && ball_dy > 0)
    {
        // extract leading edge of ball
        int x = ball_x+(BALL_SIZE/2);
        int y = ball_y+(BALL_SIZE/2);

        // test for collision with paddle
        if ((x >= paddle_x && x <= paddle_x+PADDLE_WIDTH) &&
            (y >= paddle_y && y <= paddle_y+PADDLE_HEIGHT))
        {
            // reflect ball
            ball_dy=-ball_dy;

            // push ball out of paddle since it made contact
            ball_y+=ball_dy;

            // add a little english to ball based on motion of paddle
            if (KEY_DOWN(VK_RIGHT))

```

```

        ball_dx -= (rand() % 3);
    else
    if (KEY_DOWN(VK_LEFT))
        ball_dx += (rand() % 3);
    else
        ball_dx += (-1 + rand() % 3);

    // test if there are no blocks, if so send a message
    // to game loop to start another level
    if (blocks_hit >= (NUM_BLOCK_ROWS * NUM_BLOCK_COLUMNS))
    {
        game_state = GAME_STATE_START_LEVEL;
        level++;
    } // end if

    // make a little noise
    MessageBeep(MB_OK);

    // return
    return;

} // end if

} // end if

// now scan thru all the blocks and see if ball hit blocks
for (int row=0; row < NUM_BLOCK_ROWS; row++)
{
    // reset column position
    x1 = BLOCK_ORIGIN_X;

    // scan this row of blocks
    for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
    {
        // if there is a block here then test it against ball
        if (blocks[row][col] != 0)
        {
            // test ball against bounding box of block
            if ((ball_cx > x1) && (ball_cx < x1 + BLOCK_WIDTH) &&
                (ball_cy > y1) && (ball_cy < y1 + BLOCK_HEIGHT))
            {
                // remove the block
                blocks[row][col] = 0;

                // increment global block counter, so we know
                // when to start another level up
                blocks_hit++;

                // bounce the ball

```

```
        ball_dy=-ball_dy;

        // add a little english
        ball_dx+=(-1+rand()%3);

        // make a little noise
        MessageBeep(MB_OK);

        // add some points
        score+=5*(level+(abs(ball_dx)));

        // that's it -- no more block
        return;

    } // end if

} // end if

// advance column position
x1+=BLOCK_X_GAP;
} // end for col

// advance to next row position
y1+=BLOCK_Y_GAP;

} // end for row

} // end Process_Ball

////////////////////////////////////

int Game_Main(void *parms)
{
    // this is the workhorse of your game it will be called
    // continuously in real-time this is like main() in C
    // all the calls for you game go here!

    char buffer[80]; // used to print text

    // what state is the game in?
    if (game_state == GAME_STATE_INIT)
    {
        // initialize everything here graphics
        DD_Init(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP);

        // seed the random number generator
        // so game is different each play
        srand(Start_Clock());
    }
}
```

```

    // set the paddle position here to the middle bottom
    paddle_x = PADDLE_START_X;
    paddle_y = PADDLE_START_Y;

    // set ball position and velocity
    ball_x = 8+rand()%(SCREEN_WIDTH-16);
    ball_y = BALL_START_Y;
    ball_dx = -4 + rand()%(8+1);
    ball_dy = 6 + rand()%2;

    // transition to start level state
    game_state = GAME_STATE_START_LEVEL;

    } // end if
    //////////////////////////////////////
else
if (game_state == GAME_STATE_START_LEVEL)
{
    // get a new level ready to run

    // initialize the blocks
    Init_Blocks();

    // reset block counter
    blocks_hit = 0;

    // transition to run state
    game_state = GAME_STATE_RUN;

    } // end if
    //////////////////////////////////////
else
if (game_state == GAME_STATE_RUN)
{
    // start the timing clock
    Start_Clock();

    // clear drawing surface for the next frame of animation
    Draw_Rectangle(0,0,SCREEN_WIDTH-1, SCREEN_HEIGHT-1,200);

    // move the paddle
    if (KEY_DOWN(VK_RIGHT))
    {
        // move paddle to right
        paddle_x+=8;

        // make sure paddle doesn't go off screen
        if (paddle_x > (SCREEN_WIDTH-PADDLE_WIDTH))
            paddle_x = SCREEN_WIDTH-PADDLE_WIDTH;
    }
}

```



```

    } // end if
else
if (KEY_DOWN(VK_LEFT))
{
    // move paddle to right
    paddle_x-=8;

    // make sure paddle doesn't go off screen
    if (paddle_x < 0)
        paddle_x = 0;

} // end if

// draw blocks
Draw_Blocks();

// move the ball
ball_x+=ball_dx;
ball_y+=ball_dy;

// keep ball on screen, if the ball hits the edge of
// screen then bounce it by reflecting its velocity
if (ball_x > (SCREEN_WIDTH - BALL_SIZE) || ball_x < 0)
{
    // reflect x-axis velocity
    ball_dx=-ball_dx;

    // update position
    ball_x+=ball_dx;
} // end if

// now y-axis
if (ball_y < 0)
{
    // reflect y-axis velocity
    ball_dy=-ball_dy;

    // update position
    ball_y+=ball_dy;
} // end if
else
// penalize player for missing the ball
if (ball_y > (SCREEN_HEIGHT - BALL_SIZE))
{
    // reflect y-axis velocity
    ball_dy=-ball_dy;

    // update position

```

```

    ball_y+=ball_dy;

    // minus the score
    score-=100;

    } // end if

    // next watch out for ball velocity getting out of hand
    if (ball_dx > 8) ball_dx = 8;
    else
    if (ball_dx < -8) ball_dx = -8;

    // test if ball hit any blocks or the paddle
    Process_Ball();

    // draw the paddle and shadow
    Draw_Rectangle(paddle_x-8, paddle_y+8,
                   paddle_x+PADDLE_WIDTH-8,
                   paddle_y+PADDLE_HEIGHT+8,0);

    Draw_Rectangle(paddle_x, paddle_y,
                   paddle_x+PADDLE_WIDTH,
                   paddle_y+PADDLE_HEIGHT,PADDLE_COLOR);

    // draw the ball
    Draw_Rectangle(ball_x-4, ball_y+4, ball_x+BALL_SIZE-4,
                   ball_y+BALL_SIZE+4, 0);
    Draw_Rectangle(ball_x, ball_y, ball_x+BALL_SIZE,
                   ball_y+BALL_SIZE, 255);

    // draw the info
    sprintf(buffer,"F R E A K O U T          Score %d    //
                Level %d",score,level);
    Draw_Text_GDI(buffer, 8,SCREEN_HEIGHT-16, 127);

    // flip the surfaces
    DD_Flip();

    // sync to 33ish fps
    Wait_Clock(30);

    // check of user is trying to exit
    if (KEY_DOWN(VK_ESCAPE))
    {
        // send message to windows to exit
        PostMessage(main_window_handle, WM_DESTROY,0,0);

        // set exit state
        game_state = GAME_STATE_SHUTDOWN;
    }

```

```

        } // end if

        } // end if
        //////////////////////////////////////
    else
    if (game_state == GAME_STATE_SHUTDOWN)
    {
        // in this state shut everything down and release resources
        DD_Shutdown();

        // switch to exit state
        game_state = GAME_STATE_EXIT;

    } // end if

    // return success
    return(1);

} // end Game_Main

////////////////////////////////////

```

哈哈，酷吧？这是一个完整的 Win32/DirectX 游戏，表现接近上佳水平了。BLACKOUT.CPP 源文件中有几百行代码，但是我们只能将之视为类 DirectX 游戏而且是由某个人编写的（我！）。不管怎样说，还是让我们迅速浏览一下实例 1.3 的内容吧。

首先，Windows 需要一个事件循环。这就是 Windows 编程的标准，因为 Windows 在大部分情况下都是事件驱动的。但是游戏却不是事件驱动的，无论用户在干什么，它们都在一直运行。因此，我们至少需要支持小型事件循环以取悦于 Windows。执行这项功能的代码位于 WinMain() 中——呀，令人惊奇！不是吗？

WinMain() 是所有 Windows 程序的主要入口点，就像是 Main() 是所有 DOS/UNIX 程序中的入口点一样。任何情况下，FREAKOUT 的 WinMain() 创建一个窗口，正确进入到事件循环中。如果 Windows 需要工作时，就按照这样做。当所有的基本事件处理都结束时，访问 Game_Main()。这就是我们的游戏程序实际运行的状态。

如果需要的话，读者可以一直在 Game_Main() 中循环，而不释放回到 WinMain() 主事件循环体中。但这样做不是件好事，因为 Windows 会得不到任何信息，从而缺乏资源。哎，我们想要做的是运行一个动画和逻辑的画面，然后返回到 WinMain()。这样的话，Windows 可以继续工作和处理信息。如果所有这些听起来像是幻术的话，请不要担心——在下一章中情况还会更糟。

一旦进入 Game_Main()，就执行 FreakOut 逻辑。游戏图像提交到显示缓冲区，最后通过 DD_FLIP() 访问在循环结束时在显示屏上显示出来。因此我希望读者要做的是浏览一下全部的游戏状态，跟随一遍游戏循环的每一部分，了解一下工作原理。要想玩游戏的话，

只要单击 **FREAKOUT.EXE** 文件，程序就会立即启动。控制部分包括：

右箭头键——向右移动操作杆。

左箭头键——向左移动操作杆。

Esc 键——结束并返回到 Windows。

还有，如果读者错过一个球的话，将被罚掉 100 点，可要仔细盯紧啊！

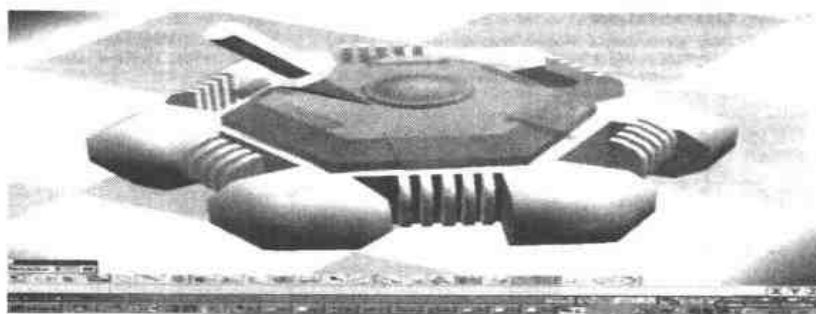
如果读者对游戏代码和玩法已适应，试着修改一下游戏。可以增加不同的背景颜色（0~255 是各种有效的颜色）、更多的球、可以改变操作杆的大小以及更多的声音效果（那是我用 Win32 API 功能 **MessageBeep()** 函数做出的）。

总 结

这大概是我所介绍的关于游戏编程快速入门课程中 fastest 的一次了！我们谈论了大量的基础内容，但是只能把它看作本书的基础版本。我只想让读者对本书中我们所讨论的和学习的内容有一个感性认识。另外，阅读一个完整的游戏是有益的，因为这可以让读者思考若干问题。

现在，在进入关于 Windows 编程的第二章之前，读者对编译 **FreakOut** 游戏应该能够轻松驾驭。如果还没有这种感觉的话，就请立即打开编译器的书和 **RTFM**，我等着你们。

2



Windows 编程模型

Windows 编程就像去见牙科医生：明知道对你有好处，但就是没有人乐意去。是不是这样？在本章中，我将要使用“禅”的方法——或者换句话说，就是深入浅出地向你介绍 Windows 编程基础。我可不能保证在阅读了本章后你就会“去见牙科医生”，但是我敢保证你会比以往更喜欢 Windows 编程。下面是本章的内容：

- Windows 的历史
- Windows 的基本风格
- Windows 的类
- 创建 Windows
- Windows 事件句柄
- 事件驱动编程和事件循环
- 打开多个窗口

Windows 的历史

读者可能会因为我要解放你的思想而感到十分恐惧（特别是钟情于 DOS 的顽固分子）。让我们迅速浏览一下 Windows 的发展历程以及与游戏发展的关系，好吗？

早期的 Windows 版本

Windows 的发展始于 Windows 1.0 版本。这是 Microsoft 公司在商业视窗操作系统的第

一次尝试，当然是一个非常失败的产品。Windows 1.0 完全建立在 DOS 基础上（这就是一个错误），不能执行多任务，运行速度很慢，看上去也差劲。它的外观可能是其失败的最重要原因。除了讽刺以外，问题还在于 Windows 1.0 与那个时代的 80286 计算机（或更差的 8086）所能提供的相比需要更高的硬件、图像和声音性能。

然而，Microsoft 稳步前进，很快就推出了 Windows 2.0。我记得获得 Windows 2.0 的测试版时我正在软件出版公司工作。在会议室中，挤满了公司的行政官员和董事长（像往常一样，他正拿着一杯鸡尾酒）。我们运行 Windows 2.0 测试演示版，装载了多个应用程序，看上去似乎还在工作。但是，那时 IBM 推出了 PM。PM 看上去要好得多，它是建立在比 Windows 2.0 先进得多的操作系统 OS/2 的基础上的，而 Windows 2.0 依然是建立在 DOS 基础上的视窗管理器。那天会议室中的结论是“不错，但还不是一个可行的操作系统，如果我们仍然留恋在 DOS 上，那我还能有鸡尾酒喝吗？”

Windows 3.x

1990 年，终于发生了翻天覆地的变化，因为 Windows 3.0 出世了，而且其表现的确非常出色！尽管它仍然赶不上 Mac OS 的标准，但是谁还在意呢？（真正的程序员都憎恨 Mac）。软件开发人员终于可以在 PC 机上创建迷人的应用程序了，而商用应用程序也开始脱离 DOS。这成了 PC 机的转折点，终于将 Mac 完全排除在商用应用程序之外了，而后也将其挤出台式机出版业。（那时，Apple 公司每 5 分钟就推出一种新硬件）。

尽管 Windows 3.0 工作良好，却还是存在许多的问题、软件漏洞，但从技术上说它已是 Windows 2.0 之后的巨大突破，有问题也是在所难免。为了解决这些问题，Microsoft 推出了 Windows 3.1，开始公关部和市场部打算称之为 Windows 4.0，但是，Microsoft 决定只简单地称之为 Windows 3.1，因为它还不足以称之为升级的换代版本。它还没有做到市场部广告宣传的那样棒。

Windows 3.1 非常可靠。它带有多媒体扩展以提供音频和视频支持，而且它还是一个出色的、全面的操作系统，用户能够以统一的方式来工作。另外，还存在一些其他的版本，如可以支持网络的 Windows 3.11（适用于工作组的 Windows）。惟一的问题是 Windows 3.1 仍然是一个 DOS 应用程序，运行于 DOS 扩展器上。

Windows 95

另一方面，游戏编程行业还在唱“DOS 永存！”的赞歌，而我则已经开始热衷于使用 Windows 3.1。但是，1995 年世界开始冷却——Windows 95 终于推出。它是一个真正 32 位的、多任务、多线程的操作系统。诚然，其中还残存一些 16 位代码，但在极大程度上，Windows 95 是 PC 机的终极开发和发布平台。

（当然，Windows NT 3.0 也同时推出，但是 NT 对于大多数用户来讲还是不可用的，

因此这里也就不再赘述)。

当 Windows 95 推出后,我才真正开始喜欢 Windows 编程。我一直痛恨使用 Windows 1.0、2.0、3.0 和 3.1 来编程,尽管随着每一种版本的推出,这种憎恨都越来越少。当 Windows 95 出现时,它彻底改变了我的思想,如同其他被征服的人的感觉一样——它看上去非常酷!那正是我所需要的。

提 示

游戏编程行业中最重要的是游戏表现如何,游戏的画面如何。同时还要尽可能减轻审阅人的工作。

因此几乎是一夜间,Windows 95 就改变了整个计算机行业。的确,目前还有一些公司仍然在使用 Windows 3.1 (你能相信吗?),但是 Windows 95 使得基于 Intel 的 PC 成为除游戏之外的所有应用程序的选择。不错,尽管游戏程序员知道 DOS 退出游戏编程行业只是一个时间的问题了,但是 DOS 还是它们的核心。

1996 年,Microsoft 公司发布了 Game SDK(游戏软件开发工具包),这基本上就是 DirectX 的第一个版本。这种技术仅能在 Windows 95 环境下工作,但是它的确太慢了,甚至竞争不过 DOS 游戏(如 DOOM 和 Duke Nukem 等)。游戏开发人员继续使用 DOS32 来开发游戏,但是他们知道 DirectX 具有足够快的速度,从而能使游戏能够流畅地运行在 PC 机上已为时不远。

到了 3.0 版,DirectX 的速度在同样计算机上已经和 DOS32 一样快了。到了 5.0 版,DirectX 已经相当完善,实现了该技术最初的承诺。对此我们将在第五章“DirectX 基础和令人生畏的 COM”涉及 DirectX 时再作详细介绍。现在要意识到:Win32/DirectX 是 PC 机上开发游戏的惟一方式。这是历史的选择。

Windows 98

1998 年中期,Windows 98 推出了。这至多是一个改进的版本,而不像 Windows 95 那样是一个换代的产品,但毫无疑问它也是占有很重要的地位。Windows 98 像一辆旧车改装的高速汽车——实际上它是一头皮毛圆润光滑、脚力持久、飞奔跳动的驴。它是全 32 位的,能够支持你想做的所有事情,并具有无限扩充的能力。它很好地集成了 DirectX、3D 图形、网络以及 Internet。

Windows 98 和 Windows 95 相比也非常稳定。当然 Windows 98 仍然经常死机,但可以相信的是,这比 Windows 95 少了许多。对即插即用支持得很好,并且能够很好地运行——这只是个时间问题。

Windows NT

现在我们来讨论一下 Windows NT。在本书编写期间，Windows NT 正在推出 5.0 版本。我所能说的是，它最终将取代 Windows 9X 成为每个人的操作系统选择。NT 要比 Windows 9X 严谨得多；而且绝大多数游戏程序员都将在 NT 上开发游戏，这将使 Windows 9X 退出历史舞台。Windows NT 5.0 最酷的是它完全支持即插即用和 Win32/DirectX，因此使用 DirectX 为 Windows 9X 编写的应用程序可以在 Windows NT 5.0 或更高版本上运行。这可是个好消息，因为从历史上看，编写 PC 游戏的开发人员现在具有最大的市场潜力。

那么最低标准是什么呢？如果你使用 DirectX（或其他工具）编写了一个 Win32 应用程序，它完全可以在 Windows 95、98 和 NT 5.0 或更高版本上运行。这可是件好事情。因此你在本书中所学到的任何东西至少可以应用到三种操作系统上，也可以运行于安装 NT 和 DirectX 的其他计算机上，如 DEC 的 Alphas。还有 Windows CE——DirectX 和 Win32 衍生的运行于其他系统上的操作系统。

Windows 的基本风格：Win9X/NT

和 DOS 不同，Windows 是一个多任务的操作系统，允许许多应用程序和更小的程序同时运行，可以最大限度的发挥硬件的性能。这表明 Windows 是一个共享的环境——一个应用程序不可能独占整个系统。尽管 Windows 95、98 和 NT 很相似，但仍然存在许多技术上的差别。但是就我们所涉及的，不可能去详细归纳。这里所参照的 Windows 机器一般是指 Win9X/NT 或 Windows 环境。让我们开始吧！

多任务和多线程

如我所说，Windows 允许不同的应用程序以轮流的方式同时执行，每一个应用程序都占用一段很短的时间段来运行，下一个应用程序轮换运行。如图 2.1 所示，CPU 由几个不同的应用程序以轮流的方式共享。判断出下一个运行的应用程序、分配给每个应用程序的时间量是调度程序的工作。

调度程序可以非常简单——每个应用程序分配固定的运行时间，也可以非常复杂——将应用程序设定为不同的优先级和抢先性或低优先级的事件。就 Win9X/NT 而言，调度程序采用基于优先级的抢先占用方式。这就意味着一些应用程序要比其他的应用程序占用处理器更多的时间，但是如果一个应用程序需要 CPU 处理的话，在另一任务运行的同时，当前的任务可以被锁定或抢先占用。

但是不要对此有太多担心，除非你正在编写 OS（操作系统）或实时代码——其细节事

关重大。大多数情况下，Windows 将执行和调度你的应用程序，无需你参与。

深入接触 Windows，我们可以看到，它不仅是多任务的，而且还是多线程的。这意味着程序由许多更简单的多个执行线程构成。这些线程（像更重要的进程）如程序一样被调度。实际上，在你的计算机上可同时运行 30~50 个线程，执行不同的任务。所以事实上你可能只运行一个程序，但这个程序由一个或多个执行线程构成。

Windows 实际的多线程示意图如图 2.2 所示，从图中可以看到，每一个程序实际上都是由一个主线程和几个工作线程构成。

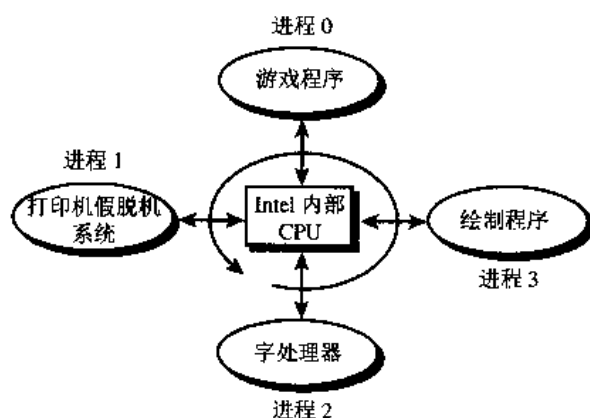


图 2.1 一个处理器执行多个程序的操作

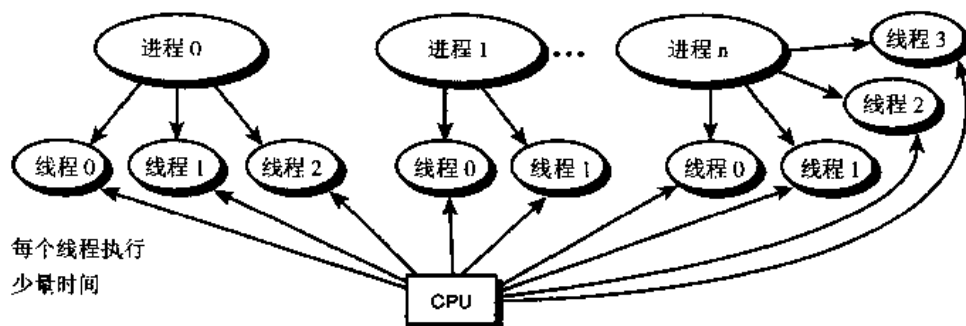


图 2.2 Windows 的更实际的多流程示意图

获取线程的信息

下面让我们来看一下你的计算机现在正在运行多少个线程。在 Windows 机器上，同时按 **Ctrl+Alt+Delete** 键，弹出显示正在运行的任务（过程）的当前程序任务管理器。这和我们所希望的不同，但也很接近。我们希望的是一个显示正在执行的实际线程数的工具或程

序。许多共享软件和商用软件工具都能做到这一点，但是 Windows 内嵌了这几个工具。

在安装 Windows 的目录（一般是 Windows\）下，可以发现一个名字为 SYSMON.EXE（Windows 95/98）或 PREFMON.EXE（Windows NT）的可执行程序。图 2.3 描述了在我的 Windows 98 机器上运行的 SYSMON.EXE 程序。图中除了正在运行的线程外还有大量的信息，如：内存使用和处理器的装载等。实际上在进行程序开发时，我喜欢使 SYSMON.EXE 运行，由此可以了解正在进行什么以及系统如何加载程序。

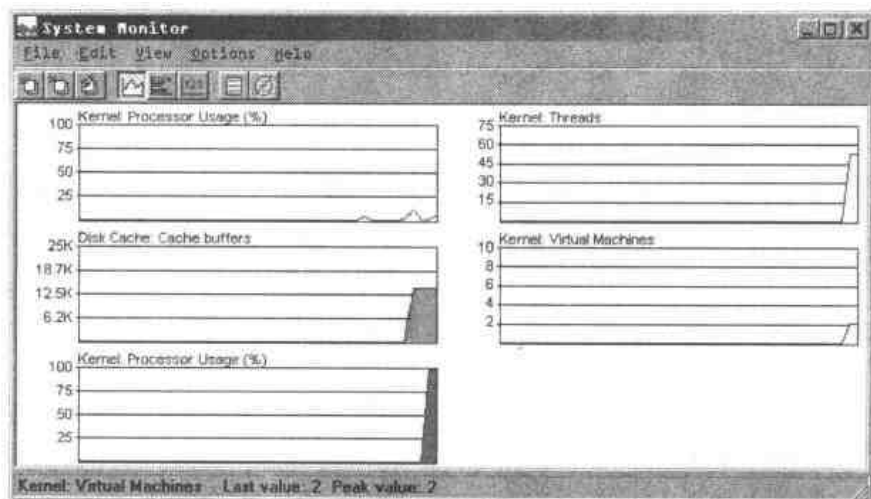


图 2.3 运行的 SYSMON.EXE

你可能想知道能否对线程的创建进行控制，答案是能够!!! 实际上这是 Windows 游戏编程最令人激动的事情之一——就像我们所希望的那样除了游戏主进程外，还能够执行其他的任务，我们也能够创建像其他任务一样多的线程。

注意

在 Windows 98/NT 环境下，实际上还有一种叫 fiber 的新型执行对象，它比线程还简单（明白吗？线程是由 fiber 构成的）。

这和 DOS 游戏程序的编写有很大不同。DOS 是单线程操作系统，也就是说一旦你的程序开始运行，就只能运行该程序（不时出现的中断管理除外）。因此，如果想使用任何一种多任务或多线程，就必须自己来模拟（参阅《Sams Teach Yourself Game Programming in 21 Days》中关于一个完整的基于 DOS 的多任务核心部分）。这也正是游戏程序员在这么多年中所作的事。的确，模拟多任务和多线程远远不能和拥有一个完整的支持多任务和多线程的操作系统相提并论，但是对于单个游戏来讲，它足可以良好地工作。

在我们接触到真正的 Windows 编程和那些工作代码之前，我想提及一个细节。你可能在想，Windows 真是一个神奇的操作系统，因为它允许多个任务和程序立即执行。请记住，实际上并不是这样的。如果只有一个处理器的话，那么一次也只能执行一个执行流、线程、程序或你所调用的任何对象。Windows 相互之间的切换太快了，以至于看上去就像几个程

序在同时运行一样。另一方面，如果有几个处理器的话，可以同时运行多个程序。例如，我有一个双 CPU 的 Pentium II 计算机，有两个 400MHz 的 Pentium II 处理器在运行 Windows NT 5.0。使用这种配置，可以同时执行两个指令流。

我希望在不远的将来，个人计算机的新型微处理器结构能够允许多个线程或 fiber 同时执行，将这样一个目标作为处理器设计的一部分。例如，Pentium 具有两个执行单元——U 管和 V 管。因此它能够立即执行两个指令。但是，这两个指令都是来自同一个线程。类似的是 Pentium II 能够立即执行 5 个简单的指令，但也是来自同一个线程。

事件模型

Windows 是个多任务/多线程 的操作系统，并且还是一个事件驱动的操作系统。和 DOS 程序不同的是，Windows 程序都是等着用户去使用，由此而触发一个事件，然后 Windows 对该事件发生响应，进行动作。请看图 2.4 所示的示意图，图中描述了大量的应用程序窗口，每个程序都向 Windows 发送待处理的事件和消息。Windows 对其中的一些进行处理，大部分的消息和事件被传递给应用程序来处理。

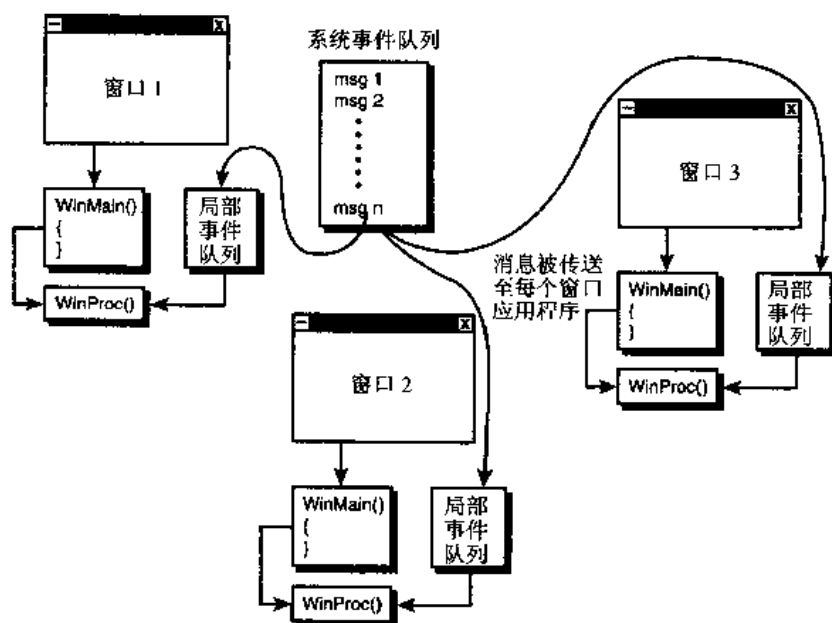


图 2.4 Windows 事件管理程序

这样做的好处是你不必去关心其他的正在运行的应用程序，Windows 会为你处理它们。你所要关心的就是你自己的应用程序和窗口中信息的处理。这在 Windows 3.0/3.1 中是根本不可能的。Windows 的那些版本并不是真正的多任务操作系统，每一个应用程序都要产生下一个程序。也就是说，在这些版本的 Windows 下运行的应用程序感觉相当粗糙、缓慢。如果有其他应用程序干扰系统的话，这个正在“温顺地”运行的程序将停止工作。但这种

情况在 Windows 9X/NT 下就不会出现。操作系统将会在适当的时间终止你的应用程序——当然，运行速度非常快，你根本就不会注意到。

到现在为止，读者已了解了所有有关操作系统的概念。幸运的是，有了 Windows 这个目前最好的编写游戏的操作系统，读者根本就不必担心程序调度——你所要考虑的就是游戏代码和如何最大限度地发挥计算机的性能。

在本章后面内容中，我们要接触一些实际的编程工作，便于读者了解 Windows 编程有多容易。但是（永远都有但是）在进行实际编程之前，我们应当了解一些 Microsoft 程序员喜欢使用的约定。这样你就不会被那些古怪的函数和变量名弄得不知所措。

按照 Microsoft 方式编程：匈牙利符号表示法

如果你正在运作一个像 Microsoft 一样的公司，有几千个程序员在干不同的项目，在某一点上就应当提出一个编写代码的标准方式。否则，结果将是一片混乱。因此一个名字叫 Charles Simonyi 的人被委托创立了一套编写 Microsoft 代码的规范。这个规范已经用作编写代码的基本指导说明书。所有 Microsoft 的 API、界面、技术文件等等都采用这些规范。

这个规范通常被称为匈牙利符号表示法，可能是因为创立这个规范工作花了很长时间，弄得他饥肠辘辘的原因吧（英文中饥饿和匈牙利谐音），或者可能他是匈牙利人。对此我们根本就不知道，关键是你必须了解这个规范，以便于你能够阅读 Microsoft 代码。

匈牙利符号表示法包括许多与下列命名有关的约定：

- 变量
- 函数
- 类型和常量
- 类
- 参数

表 2.1 给出了匈牙利符号表示法使用的前缀代码。这些代码在大多数情况下一半用于前缀变量名，其他约定根据名称确定。其他解释可以参考本表。

表 2.1 匈牙利符号表示法的前缀代码指导说明书

前 缀	数据类型（基本类型）
c	字符
by	字节（无符号字符）
n	短整数和整数（表示一个数）
i	整数
x, y	短整数（通常用于 x 坐标和 y 坐标）

续表

前 缀	数据类型（基本类型）
cx, cy	短整数（通常用于表示 x 和 y 的长度；c 表示计数）
b	布尔型（整数）
w	UINT（无符号整数）和 WORD（无符号字）
l	LONG（长整数）
dw	DWORD（无符号长整数）
fn	函数指针
s	串
sz, str	以 0 字节终止的字符串
lp	32 位的长整数指针
h	编号（常用于表示 Windows 对象）
msg	消息

变量的命名

应用匈牙利符号表示法，变量可用表 2.1 中的前缀代码来表示。另外，当一个变量是由一个或几个子名构成时，每一个子名都要以大写字母开头。下面是几个例子：

```
char *szFileName;    // a null-terminated string
int *lpDate;         // a 32-bit pointer to an int
BOOL bSemaphore;     // a boolean value
WORD dwMaxCount;     // a 32-bit unsigned WORD
```

尽管我了解一个函数的局部变量没有说明，但是也有个别表示全局变量：

```
int g_iXPos;          // a global x-position
int g_iTimer;         // a global y-position
char *g_szString;     // a global NULL-terminated string
```

总的来说，变量以 g_ 开头，或者有时就只用 g。

函数的命名

函数和变量命名方式相同，但是没有前缀。换句话说，子名的第一个字母要大写。下面是几个例子：

```
int PlotPixel(int ix, int iy, int ic);
void *MemScan(char *szString);
```

而且，下划线是非法的。例如，下面的函数名表示是无效的匈牙利符号表示法：

```
int Get_Pixel(int ix, int iy);
```

类型和常量的命名

所有的类型和常量都是大写字母，但名字中可以允许使用下划线。例如：

```
const LONG NUM_SECTORS = 100;    // a C++ style constant
#define MAX_CELLS 64;            // a C style constant
#define POWERUNIT 100;           // a C style constant
typedef unsigned char UCHAR;     // a user defined type
```

这儿并没有什么不同的地方——非常标准的定义。尽管大多数 Microsoft 程序员不使用下划线，但我还是喜欢用，因为这样能使名字更具有可读性。

C++

在 C++ 中，关键字 `const` 不止一个意思。在前面的代码行中，它用来创建一个常数变量。这和 `#define` 相似，但是它增加了类型信息这个特性。`Const` 不仅仅像 `#define` 一样是一个简单的预处理文本替换，而且更像是一个变量。它允许编译器进行类型检查和替换。

类的命名

类命名的约定可能要麻烦一点。但我也看到有很多人在使用这个约定，并独立地进行补充。不管怎样说，所有 C++ 的类必须以大写 **C** 为前缀，类名字的每一个子名的第一个字母都必须大写。下面是几个例子：

```
class CVector
{
public

    CVector (); {ix=iy=iz=imagnitude = 0;}
    CVector(int x, int y, int z) {ix=x; iy=y; iz=z;}
    .
    .
private:
    int ix, iy, iz;    // the position of the vector
    int imagnitude;   // the magnitude of the vector

};
```

参数的命名

函数的参数命名和标准变量命名的约定相同。但也不总是如此。例如下面例子给出了一个函数定义：

```
UCHAR GetPixel(int x, int y);
```

这种情况下，更准确的匈牙利函数原型是：

```
UCHAR GetPixel(int ix, int iy);
```

但我认为这并没有什么两样。

最后，你甚至可能都看不到这些变量名，而仅仅看到类型，如下所示：

```
UCHAR GetPixel(int, int);
```

当然，这仅仅是原型使用的，真正的函数声明必须带有可赋值的变量名，这一点你已经掌握了。

注意



仅仅会读匈牙利符号表示法并不代表你能使用它。实际上，我进行编程工作已经有 20 多年了，我也不准备为谁改变我的编程风格。因此，本书中的代码使用类匈牙利符号表示法的编码风格，这是 Win32 API 造成的，在其他位置将使用我自己的风格。必须注意的是，我使用的变量名的第一个字母没有大写，并且我还使用下划线。

世界上最简单的 Windows 程序

现在读者已经对 Windows 操作系统及其性能和基本设计问题有了一般的了解，那就让我们从第一个 Windows 程序开始真正的 Windows 编程吧。

以每一种新语言或所学的操作系统来编写一个“世界你好”的程序是一个惯例，让我们也来试试。清单 2.1 是标准的基于 DOS 的“世界你好”程序。

程序清单 2.1 基于 DOS 的“世界你好”程序

```
// DEMO2_1.CPP - standard version
#include <stdio.h>

// main entry point for all standard DOS/console programs
void main(void)
{
    printf("\nTHERE CAN BE ONLY ONE!!!\n");
} // end main
```

现在让我们看一看用 Windows 如何编写它。

技巧



顺便说一句，如果读者想编译 DEMO2_1.CPP 的话，就应当用 VC++ 或 Borland 编译器实际创建一个调用内容的控制应用程序。这是一个类 DOS 的应用程序，只是它是 32 位的，它仅以文本模式运行，但对于检验一个想法和算法是很有用的。

总是从 WinMain() 开始

如前面所述，所有的 Windows 程序都以 WinMain() 开始，这和简单直观的 DOS 程序都以 Main() 开始一样。WinMain() 中的内容取决于你。如果你愿意的话，可以创建一个窗口、开始处理事件并在屏幕上画一些东西。另一方面，你可以调用几百个（或者是几千个）Win32 API 函数中的一个。这正是我们将要做的。

我只想在屏幕上的一个信息框中打印一点东西。这正是一个 Win32 API 函数 MessageBox() 的功能。清单 2.2 是一个完整的、可编译的 Windows 程序，该程序创建和显示了一个能够到处移动和关闭的信息框。

程序清单 2.2 第一个 Windows 程序

```
// DEMO2_2.CPP - a simple message box
#define WIN32_LEAN_AND_MEAN

#include <windows.h>          // the main windows headers
#include <windowsx.h>         // a lot of cool macros

// main entry point for all windows programs
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    // call message box api with NULL for parent window handle
    MessageBox(NULL, "THERE CAN BE ONLY ONE!!!",
               "MY FIRST WINDOWS PROGRAM",
               MB_OK | MB_ICONEXCLAMATION);

    // exit program
    return(0);

} // end WinMain
```

要编译该程序，按照下面步骤：

1. 创建新的 Win32.EXE 项目并包含 CD-ROM 上 T3DCHAP02\下的 DEMO2_2.CPP。
2. 编译和联接程序。

3. 运行! (或在 CD-ROM 上直接运行预编译版本 DEMO2_2.EXE)。

你可能会以为一个基本的 Windows 程序有几百行代码。当你编译和运行程序时, 可能会看到如图 2.5 所示的内容。



图 2.5 运行 DEMO2_2.EXE

程序剖析

现在已经有有了一个完整的 Windows 程序, 让我们一行一行地分析程序的内容。首先第一行程序是

```
#define Win32_LEAN_AND_MEAN
```

这个应稍微解释一下。创建 Windows 程序有两种方式——使用 Microsoft 基础类 (Microsoft Foundation Classes, MFC), 或者使用软件开发工具包 (Software Development Kit, SDK)。MFC 完全基于 C++ 和类, 要比以前的游戏编程所需的工具复杂得多, 功能和难度也要强大和复杂 10 倍。而 SDK 是一个可管理程序包, 可以在一到两周内学会 (至少初步学会), 并且使用了简单明了的 C 语言。因此, 我在本书中所使用的工具是 SDK。

Win32_LEAN_AND_MEAN 指示编译器 (实际上是逻辑头文件) 不包含无关的 MFC 操作。现在我们又离题了, 回来继续看程序。

之后, 下面的头文件是:

```
#include "windows.h"
#include "windowsx.h"
```

第一个引用 “windows.h” 实际上包括所有的 Windows 头文件。Windows 有许多这样的头文件, 这就有点像包含宏, 可以节省许多手工包含显式头文件的时间。

第二个引用 “windowsx.h” 是一个含有许多重要的宏和常量的头文件, 该文件可以简化 Windows 编程。

下面就到了最重要的部分——所有 Windows 应用程序的主要入口位置 WinMain():

```
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow);
```

首先，应当注意到奇怪的 WINAPI 声明符。这相当于 PASCAL 函数声明符，它强制参数从左边向右边传递，而不是像默认的 CDECL 声明符那样参数从右到左转移。但是，PASCAL 调用约定声明已经过时了，WINAPI 代替了该函数。必须使用 WinMain() 的 WINAPI 声明符；否则，将向函数返回一个不正确的参数并终止开始程序。

测试参数

下面让我们详细看一下每个参数：

- **hinstance**——该参数是一个 Windows 为你的应用程序生成的实例句柄。实例是一个用来跟踪资源的指针或数。本例中，hinstance 就像一个名字或地址一样，用来跟踪你的应用程序。
- **hprevinstance**——该参数已经不再使用了，但是在 Windows 的旧版本中，它跟踪应用程序以前的实例（换句话说，就是产生当前实例的应用程序实例）。难怪 Microsoft 要去除它，它就像一次长途跋涉——让我们为之头疼。
- **lpcmdline**——这是一个空终止字符串，和标准 C/C++ main(int argc, char **argv) 函数中的命令行参数相似。不同的是，它不是一个单独的像 argc 那样指出命令行的参数。例如，如果你创建一个名字为 TEST.EXE 的 Windows 应用程序，并且使用下面的参数运行：

```
TEST.EXE one
```

lpcmdline 将含有下面数据：

```
lpcmdline = "one two three"
```

注意，.EXE 的名字本身并不是命令行的一部分。

- **ncmdshow**——最后一整型参数在运行过程中被传递给应用程序，带有如何打开主应用程序窗口的信息。这样，用户便会拥有一点控制应用程序如何启动的能力。当然，作为一个程序员，如果想忽略它也可以，而想使用它也行。（你将参数传递给 ShowWindow()，我们又超前了！）表 2.2 列出了 ncmdshow 最常用的参数值。

表 2.2 ncmdshow 的 Windows 代码

值	功 能
SW_SHOWNORMAL	激活并显示一个窗口。如果该窗口最小化或最大化的话，Windows 将它恢复到原始尺寸和位置。当第一次显示该窗口时，应用程序将指定该标志
SW_SHOW	激活一个窗口，并按当前尺寸和位置显示
SW_HIDE	隐藏一个窗口，并激活另外一个窗口

续表

值	功 能
SW_MAXIMIZE	将指定的窗口最大化
SW_MINIMIZE	将指定的窗口最小化
SW_RESTORE	激活并显示一个窗口。如果该窗口最小化或最大化的话, Windows 将它恢复到原始尺寸和位置。当恢复为最小化窗口时, 应用程序必须指定该标志
SW_SHOWMAXIMIZED	激活 一个窗口, 并以最大化窗口显示
SW_SHOWMINIMIZED	激活 一个窗口, 并以最小化窗口显示
SW_SHOWMINNOACTIVE	以最小化窗口方式显示 一个窗口, 激活的窗口依然保持激活的状态
SW_SHOWNA	以当前状态显示一个窗口, 激活的窗口依然保持激活的状态
SW_SHOWNOACTIVATE	以上一次的窗口尺寸和位置来显示窗口, 激活的窗口依然保持激活的状态

如表 2.2 所示, `ncmdshow` 有许多设置 (目前许多值都没有意义)。实际上, 这些设置大部分都不在 `ncmdshow` 中传递。可以应用另一个函数 `ShowWindow()` 来使用它们, 该函数在一个窗口创建时就开始显示。对此我们在本章后面将进行详细讨论。

我想说的一点是, Windows 带有大量的你从未使用过的选项和标志等等, 就像 VCR 编程选项一样——越多越好, 任你使用。Windows 就是按照这种方式设计的。这将使每个人都感到满意, 这也意味着它包含了许多选项。实际上, 我们在 99% 时间内将会使用 `SW_SHOW`、`SW_SHOWNORMAL` 和 `SW_HIDE`, 但是你还要了解在 1% 的时间内会用到的其他选项。

选择一个信息框

最后让我们讨论一下 `WinMain()` 中调用 `MessageBox()` 的实际机制。`MessageBox()` 是一个 Win32 API 函数, 它替我们做某些事, 使我们不需自己去做。该函数经常以不同的图标和一个或两个按钮来显示信息。你看, 简单的信息显示在 Windows 应用程序中非常普通, 有了这样一个函数就节省了程序员半个多小时的时间, 而不必每次使用都要编写它。

`MessageBox()` 并没有什么多少功能, 但是能够在屏幕上显示一个窗口, 提出一个问题, 并且等候用户的输入。下面是 `MessageBox()` 的原型:

```
int MessageBox( HWND    hwnd,          // handle of owner window
                LPCTSTR  lptext,       // address of text in message box
                LPCTSTR  lpcaption,    // address of title of message box
                UINT      ustyle);     // style of message box
```

参数定义如下:

`hwnd`——这是信息框连接窗口的句柄。目前我们还不能谈及窗口句柄, 因此只能认为

它是信息框的父窗口。在 DEMO2_2.CPP，我们将它设置为空值 NULL，因此使用 Windows 桌面作为父窗口。

lp_{text}——这是一个包含显示文本的空值终止字符串。

lp_{caption}——这是一个包含显示文本框标题的空值终止字符串。

u_{type}——这大概是该簇参数中惟一令人激动的参数了，控制信息显示框的种类。

表 2.3 列出了几种 MessageBox() 选项（有些删减）。

表 2.3 MessageBox() 选项

标 志	描 述
下列设置控制信息框的一般类型	
MB_OK	信息框含有一个按钮：OK，这是默认值
MB_OKCANCEL	信息框含有两个按钮：OK 和 Cancel
MB_RETRYCANCEL	信息框含有两个按钮：Retry 和 Cancel
MB_YESNO	信息框含有两个按钮：Yes 和 No
MB_YESNOCANCEL	信息框含有三个按钮：Yes、No 和 Cancel
MB_ABORTRETRYIGNORE	信息框含有三个按钮：Yes、No 和 Cancel
这一组控制在图标上添加一点“穷人的多媒体”	
MB_ICONEXCLAMATION	信息框显示一个惊叹号图标
MB_ICONINFORMATION	信息框显示一个由圆圈中的小写字母 I 构成的图标
MB_ICONQUESTION	信息框显示一个问号图标
MB_ICONSTOP	信息框显示一个终止符图标
该标志组控制默认时高亮的按钮	
MB_DEFBUTTON _n	其中 n 是一个指示默认按钮的数字（1~4），从左到右计数

注意：还有其他的高级 OS 级标志，我们没有讨论。如果希望了解更多细节的话，可以通过编译器 Win32 SDK 的在线帮助来查阅。

可以同时使用表 2.3 中的值进行逻辑或运算，来创建一个信息框。一般情况下，只能从每一组中仅使用一个标志来进行或运算。

当然，和所有 Win32 API 函数一样，MessageBox() 函数返回一个值来通知编程者所发生的事件。但在这个例子中谁关心这个呢？通常情况下，如果信息框是 yes/no 提问之类的话，就希望知道这个返回值。表 2.4 列出了可能的返回值。

表 2.4 MessageBox() 的返回值

值	按钮选择
IDABORT	Abort
IDCANCEL	Cancel
IDIGNORE	Ignore
IDNO	No
IDOK	OK
IDRETRY	Retry
IDYES	Yes

最后，这个表已经毫无遗漏地列出了所有的返回值。现在已经完成了对我们第一个 Windows 程序——单击的逐行分析。

技巧



现在希望你能轻松地对这个程序进行修改，并以不同的方式进行编译。使用不同的编译器选项，例如优化。然后尝试通过调试程序来运行该程序，看看你是否已经领会。做完后，请回到此处。

如果希望听到声音的话，一个简单的技巧就是使用 MessageBeep() 函数，可以在 Win32 SDK 中查阅，它和 MessageBox() 函数一样简单好用。下面就是该函数原型：

```
BOOL MessageBeep(UINT uType); // 运行声音
```

可以从表 2.5 所示常数中得到不同的声音。

表 2.5 MessageBeep() 函数的声音标识符

值	声 音
MB_ICONASTERISK	系统星号
MB_ICONEXCLAMATION	系统惊叹号
MB_ICONHAND	系统指针
MB_ICONQUESTION	系统问号
MB_OK	系统默认值
0xFFFFFFFF	使用计算机扬声器的标准嘟嘟声，令人讨厌

注意：如果已经安装了 MS-Plus 主题曲的话，你应能得到有意思的结果。

看 Win32 API 多酷啊！可以有上百个函数使用。它们虽然不是世界上最快的函数，但是对于一般的内部管理 I/O 和 GUI 来讲，它们已经很棒了。

让我们稍微花点时间总结一下我们目前所知的有关 Windows 编程方面的知识。首先，Windows 支持多任务/多线程，因此可以同时运行多个应用程序。我们不必费心就可以做到这一点。我们最关心的是 Windows 支持事件触发。这就意味着我们必须处理事件（在这一点上目前我们还不知如何做）并且做出反应。好，听上去不错。最后所有 Windows 程序都以函数 WinMain() 开始，WinMain() 函数中的参数要比标准 DOS Main() 多得多，但这些参数

都属于逻辑和推理的领域。

掌握了上述的内容，就到了编写一个真正的 Windows 应用程序的时候了。

真实的 Windows 应用程序

尽管本书的目标是编写在 Windows 环境下运行的 3D 游戏，但是你并不需要了解更多的 Windows 编程。实际上，你所需要的就是一个基本的 Windows 程序，可以打开一个窗口、处理信息、调用主游戏循环等等。了解了这些，本章中的目标是首先向你展示如何创建简单的 Windows 应用程序，同时为编写类似 32 位 DOS 环境的游戏外壳程序奠定基础。

一个 Windows 程序的关键就是打开一个窗口。一个窗口就是一个显示文本和图形信息的工作区。要创建一个完全实用的 Windows 程序，只要进行下列工作：

1. 创建一个 Windows 类。
2. 创建一个事件句柄或 WinProc。
3. 用 Windows 注册 Windows 类。
4. 用前面创建的 Windows 类创建一个窗口。
5. 创建一个能够从事件句柄获得或向事件句柄传递 Windows 信息的主事件循环。

让我们详细了解一下每一步的工作。

Windows 类

Windows 实际上是一个面向对象的操作系统，因此 Windows 中大量的概念和程序都出自 C++。其中一个概念就是 Windows 类。Windows 中的每一个窗口、控件、列表框、对话框和小部件等等实际上都是一个窗口。区别它们的就是定义它们的类。一个 Windows 类就是 Windows 能够操作的一个窗口类型的描述。

有许多预定义的 Windows 类，如按钮、列表框、文件选择器等等。你也可以自己任意创建你的 Windows 类。实际上，你可以为自己编写的每一个应用程序创建至少一个 Windows 类。否则你的程序将非常麻烦。因此你应当在画一个窗口时，考虑一个 Windows 类来作为 Windows 的一个模板，以便于在其中处理信息。

控制 Windows 类信息的数据结构有两个：WNDCLASS 和 WNDCLASSEX。WNDCLASS 是比较古老的一个，可能不久将废弃，因此我们应当使用新的扩展版 WNDCLASSEX。二者结构非常相似，如果有兴趣的话，可以在 Win32 帮助中查阅 WNDCLASS。让我们看一下在 Windows 头文件中定义的 WNDCLASSEX。

```
typedef struct _WNDCLASSEX
```

```
{
UINT      cbSize;           // size of this structure
UINT      style;           // style flags
WNDPROC   lpfnWndProc;     // function pointer to handler
int        cbClsExtra;     // extra class info
int        cbWndExtra;     // extra window info
HANDLE    hInstance;      // the instance of the application
HICON     hIcon;          // the main icon
HCURSOR   hCursor;        // the cursor for the window
HBRUSH     hbrBackground; // the background brush to paint the window
LPCTSTR   lpstrMenuName;  // the name of the menu to attach
LPCTSTR   lpstrClassName; // the name of the class itself
HICON     hIconSm;        // the handle of the small icon
} WNDCLASSEX
```

因此你所要做的就是创建一个这样的结构，然后填写所有的字段：

```
WNDCLASSEX winclass; // a blank windows class
```

第一个字段 `cbSize` 非常重要（但 Petzold 在《Programming Windows 95》中忘记了这内容），它是 `WNDCLASSEX` 结构本身的大小。你可能要问，为什么应当知道该结构的大小？这个问题问得好，原因是如果这个结构作为一个指针被传递的话，接收器首先检查第一个字段，以确定该数据块最低限度有多大。这有点像提示和帮助信息，以便于其他函数在运行时不必计算该类的大小。因此，我们应当这样做：

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

第二个字段包含描述该窗口一般属性的结构信息标志。有许多这样的标志，因此我们不能全部列出它们。只要能够使用它们创建任何类型的窗口就行了。表 2.6 列出了常用的标志。读者可以任意对这些值进行逻辑“或”运算，来派生所希望的窗口类型。

表 2.6 Windows 类的类型标志

标 志	说 明
CS_HREDRAW	若移动或改变了窗口宽度，则刷新整个窗口
CS_VREDRAW	若移动或改变了窗口高度，则刷新整个窗口
CS_OWNDC	为该类中每个窗口分配一个单值的设备描述表（在本章后面详细描述）
CS_DBLCLKS	当用户双击鼠标时向窗口程序发送一个双击的信息，同时，光标位于属于该类的窗口中
CS_PARENTDC	在母窗口中设定一个子窗口的剪切区，以便于子窗口能够画在母窗口中
CS_SAVEBITS	在一个窗口中保存用户图像，以便于在该窗口被遮住、移动时不必每次刷新屏幕。但是，这样会占用更多的内存，并且比人工同样操作要慢得多
CS_NOCLOSE	禁止系统菜单上的关闭命令

注意：用黑体显示的部分为最常用的标志。

表 2.6 包含了大量的标志,即使读者对此尚有疑问,也无关紧要。现在,设定类型标识符,描述如果窗口移动或改变尺寸就进行屏幕刷新,并可以获得一个静态的设备描述表以及处理双击事件的能力。

我们将在第三章“高级 Windows 编程”中详细讨论设备描述表,但基本说来,它被用作窗口中图像着色的数据结构。因此,如果你要处理一个图像,就应为感兴趣的特定窗口申请一个设备描述表。如果设定了一个 Windows 类,它就通过 CS_OWNDNC 得到了一个设备描述表,如果你不想每次处理图像时都申请设备描述表,可以将它保存一段时间。上面说的对你有帮助还是使你更糊涂? Windows 就是这样——你知道得越多,问题就越多。好了!下面说一下如何设定类型字段:

```
winclass.style = CS_VREDRAW | CS_HREDRAW | CS_OWNDNC | CS_DBLCLKS;
```

WNDCLASSEX 结构的下一个字段 lpfnWndProc 是一个指向事件句柄的函数指针。基本上这里所设定的都是该类的回调函数。回调函数在 Windows 编程中经常使用,工作原理如下:当有事件发生时,Windows 通过调用一个你已经提供的回调函数来通知你,这省去你盲目查询的麻烦。随后在回调函数中,再进行所需的操作。

这个过程就是基本的 Windows 事件循环和事件句柄的操作过程。向 Windows 类申请一个回调函数(当然需要使用特定的原型)。当一个事件发生时,Windows 按如图 2.6 所示的那样替你调用它。关于该内容我们将在下面部分进行更详细的介绍。但是现在,读者只要将其设定到你将编写的事件函数中去:

```
winclass.lpfnWndProc = WinProc; // this is our function
```

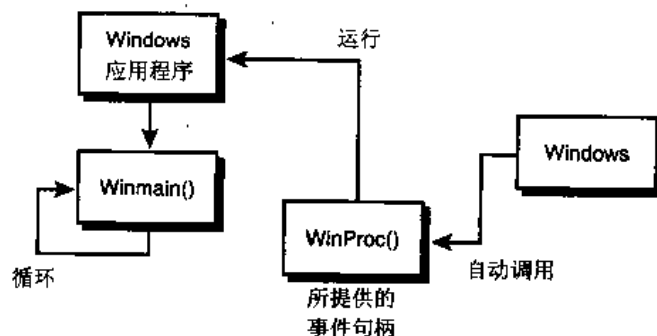


图 2.6 Windows 事件句柄回调函数的操作流程

C++

函数指针有点像 C++ 中的虚函数。如果你对它们不熟悉的话,在这里讲一下。假设有两个函数用以操作两个数:

```
int Add(int op1,int op2) {return(op1+op2);}
int Sub(int op1,int op2) {return(op1-op2);}
```


C++

要想用同一调用来调用两个函数，可以用一个函数指针来实现。如下：

```
// define a function pointer that takes two int and returns
an int
int (Math*)(int, int);
```

然后可以如下配置函数指针：

```
Math = Add;
int result = Math(1,2); // this really calls Add(1,2)
// result will be 3
```

```
Math = Sub;
int result = Math(1,2); // this really calls Sub(1,2)
// result will be -1
```

看，不错吧。

下面两个字段，`cbClsExtra` 和 `cbWndExtra` 原是为指示 Windows 将附加的运行时间信息保存到 Windows 类某些单元中而设计的。但是绝大多数人使用这些字段并简单地将其值设为 0，如下所示：

```
winclass.cbClsExtra = 0; // extra class info space
winclass.cbWndExtra = 0; // extra window info space
```

下一个是 `hInstance` 字段。这是一个简单的、在启动时传递给 `WinMain()` 函数的句柄实例，因此只需简单地从 `WinMain()` 中复制即可：

```
winclass.hInstance = hinstance; // assign the application instance
```

剩下的字段和 Windows 类的图像方面有关，在讨论它们之前，先花一点时间回顾一下句柄。

在 Windows 程序和类型中将一再看到句柄：位图句柄、光标句柄、任意事情的句柄。请记住，句柄只是一个基于内部 Windows 类型的标识符。其实它们都是整数。但是 Microsoft 可能改变这一点，因此安全使用 Microsoft 类型是个好主意。总之，你将会看到越来越多的“[...] 句柄”，请记住，有前缀 `h` 的任何类型通常都是一个句柄。好，回到原来的地方继续吧。

下一个字段是设定表示应用程序的图标的类型。你完全可以装载一个你自己定制的图标，但现在你使用系统图标，需要为它设置一个句柄。要为一个常用的系统图标检索一个句柄，可以使用 `LoadIcon()` 函数：

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

这行代码装载一个标准的应用程序图标——虽然烦人，但是简单。如果对 `LoadIcon()` 函数有兴趣的话，请看下面的它的原型，表 2.7 给出了几个图标选项：

```
HICON LoadIcon(HINSTANCE hInstance, // handle of application instance
    LPCTSTR lpIconName); // icon-name string or icon resource identifier
```

`hInstance` 是一个从应用程序装载图标资源的实例（后面将详细讨论）。现在将它设置为 `NULL` 来装载一个标准的图标。`LpIconName` 是包含被装载图标资源名称的 `NULL` 终止字符串。当 `hInstance` 为 `NULL` 时，`lpIconName` 的值如表 2.7 所示。

表 2.7 `LoadIcon()` 的图标标识符

值	说 明
<code>IDI_APPLICATION</code>	默认应用程序图标
<code>IDI_ASTERISK</code>	星号
<code>IDI_EXCLAMATION</code>	惊叹号
<code>IDI_HAND</code>	手形图标
<code>IDI_QUESTION</code>	问号
<code>IDI_WINLOGO</code>	Windows 徽标

好，现在我们已介绍了一半的字段了。做个深呼吸休息一会，让我们进行下一个字段 `hCursor` 的介绍。和 `hIcon` 相似，它也是一个图像对象句柄。不同的是，`hCursor` 是一个指针进入到窗口的用户区才显示的光标句柄。使用 `LoadCursor()` 函数可以得到资源或预定义的系统光标。我们将在后面讨论资源，简单而言资源就是像位图、光标、图标、声音等一样的数据段，它被编译到应用程序中并可以在运行时进行访问。Windows 类的光标设定如下所示：

```
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

下面是 `LoadCursor()` 函数的原型（表 2.8 列出了不同的系统光标标识符）：

```
HCURSOR lpCursor( HINSTANCE hInstance, // handle of application instance
LPCTSTR lpCursorName); // icon_name string or icon resource identifier
```

`hInstance` 是你的 .EXE 的应用程序实例。该 .EXE 应用程序包含订制光标名称来源的资源。但现在读者还不能使用该功能，仅将默认的系统光标值设定为 `NULL`。

`lpCursorName` 标识了资源名字字符串或资源句柄（我们一般不使用），或者是一个常数，以标识如表 2.8 中所示的系统默认值。

表 2.8 `LoadCursor()` 的值

值	说 明
<code>IDC_ARROW</code>	标准箭头
<code>IDC_APPSTARTING</code>	标准箭头标和小沙漏标
<code>IDC_CROSS</code>	横标线
<code>IDC_IBEAM</code>	文本 I 型标
<code>IDC_NO</code>	带正斜线的圆圈

续表

值	说 明
IDC_SIZEALL	四向箭头
IDC_SIZENESW	指向东北—西南方向的双向箭头
IDC_SIZENS	指向南北方向的双向箭头
IDC_SIZENWSE	指向东南—西北方向的双向箭头
IDC_SIZEWE	指向东西方向的双向箭头
IDC_UPARROW	垂直方向的箭头
IDC_WAIT	沙漏

现在我们要解放了，因为我们几乎已经全部介绍完了——剩下的字段更有意义。让我们看一看 `hbrBackground`。

无论在什么时候绘制或刷新一个窗口，Windows 都至少将以用户预定义的颜色或 Windows 内部设置的画笔颜色填充该窗口的背景。因此，`hbrbackground` 是一个用于窗口刷新的画笔句柄。画笔、笔、色彩和图形都是 GDI（图形设备接口）的一部分，我们将在下一章中详细讨论。现在，介绍一下如何申请一个基本的系统画笔来填充窗口。该项功能由 `GetStockObject()` 来实现，如下面程序所示：

```
winclass.hbrBackground = GetStockObject(WHITE_BRUSH);
```

`GetStockObject()` 是一个通用函数，用于获得 Windows 系统画笔、笔、调色板或字体的一个句柄。`GetStockObject()` 只有一个参数，用来指示装载哪一项资源。表 2.9 仅列出了画笔和笔的可能库存对象。

表 2.9 `GetStockObject()` 的库存对象标识符

值	说 明
BLACK_BRUSH	黑色画笔
WHITE_BRUSH	白色画笔
GRAY_BRUSH	灰色画笔
LTGRAY_BRUSH	淡灰色画笔
DKGRAY_BRUSH	深灰色画笔
HOLLOW_BRUSH	空心画笔
NULL_BRUSH	无效（NULL）画笔
BLACK_PEN	黑色笔
WHITE_PEN	白色笔
NULL_PEN	无效（NULL）笔

WNDCLASS 结构中的下一个字段是 lpszMenuName。它是菜单资源名称的空终止 ASCII 字符串，用于加载和选用窗口。其工作原理将在第三章“高级 Windows 编程”中讨论。现在我们只需将值设为 NULL：

```
Winclass.lpszMenuName=NULL; //the name of the menu to attach
```

如我刚提及的那样，每个 Windows 类代表你的应用程序所创建的不同窗口类型。在某种程度上，类与模板相似，Windows 需要一些途径来跟踪和识别它们。因此，下一个字段 lpszClassName，就用于该目的。该字段被赋以包含相关类的文本标示符的空终止字符串。我个人喜欢用诸如“WINCLASS1”、“WINCLASS2”等标示符。读者以自己喜好而定，以简单明了为原则，如下所示：

```
Winclass.lpszClassName = "WINCLASS1"; // the name of the class itself
```

这样赋值以后，你可以使用它的名字来引用这个新的 Windows 类——很酷，是吗？

最后就是小应用程序图标。这是 Windows 类 WNDCLASSEX 中新增加的功能，在老版本 WNDCLASS 中没有。首先，它是指向你的窗口标题栏和 Windows 桌面任务栏的句柄。你经常需要装载一个自定义资源，但是现在只要通过 LoadIcon() 使用一个标准的 Windows 图标即可实现：

```
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION); // 小图标句柄
```

下面让我们迅速回顾一下整个类的定义：

```
WNDCLASSEX winclass; // this will hold the class we create

// first fill in the window class structure
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = "WINCLASS";
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

当然，如果想节省一些打字时间的话，可以像下面这样简单地初始化该结构：

```
WNDCLASSEX winclass = {
    winclass.cbSize = sizeof(WNDCLASSEX),
    CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW,
    WindowProc,
```

```

0,
0,
hinstance,
LoadIcon(NULL, IDI_APPLICATION),
LoadCursor(NULL, IDC_ARROW),
GetStockObject(BLACK_BRUSH),
NULL,
"WINCLASS1",
LoadIcon(NULL, IDI_APPLICATION));

```

这样就省去了许多输入！

注册 Windows 类

现在 Windows 类已经定义并且存放在 winclass 中，必须将新的类通知 Windows。该功能通过 RegisterClassEx() 函数，使用一个指向新类定义的指针来完成，如下所示：

```
RegisterClassEx( &winclass );
```



注意 我并没有使用我们例子中的“WINCLASS1”的类名。对于 RegisterClassEx() 来讲，必须使用保存该类的实际结构。因为在该类调用 RegisterClassEx() 函数之前，Windows 并不知道该类的存在。明白了吧？

此外还有一个旧版本的 RegisterClass() 函数，用于注册基于旧结构 WNDCLASS 基础上的类。

该类一旦注册，我们就可以任意创建它的窗口。请看下面如何进行这个工作，然后再详细看一下事件句柄和主事件循环，了解使一个 Windows 应用程序运行还要做哪些工作。

创建窗口

要创建一个窗口（或者一个类窗口的对象），使用 CreateWindow() 或 CreateWindowEx() 函数。后者是更新一点的版本，支持附加类型参数，我们就使用它。该函数是创建 Windows 类的函数，我们要多花一点时间来逐行分析。在创建一个窗口时，必须为这个 Windows 类提供一个正文名——我们现在就使用“WINCLASS1”命名。这是识别该 Windows 类并区别于其他类以及内嵌的诸如按钮、文本框等类型的标示。

下面是 CreateWindowEx() 函数的原型：

```

HWND CreateWindowEx(
    DWORD dwExStyle,          // extended window style

```

```

LPCTSTR lpClassName,    // pointer to registered class name
LPCTSTR lpWindowName,   // pointer to window name
DWORD dwStyle,           // window style
int x,                   // horizontal position of window
int y,                   // vertical position of window
int nWidth,              // window width
int nHeight,             // window height
HWND hWndParent,         // handle to parent or owner window
HMENU hMenu,             // handle to menu, or child-window identifier
HINSTANCE hInstance,     // handle to application instance
LPVOID lpParam);        // pointer to window-creation data

```

如果该函数执行正确的话，将返回一个指向新建窗口的句柄；否则就返回空值 NULL。

上述大多数参数是不需要加以说明的，现在让我们浏览一下：

- **dwExStyle**——该扩展类型标志具有高级特征，大多数情况下，可以设为 NULL。如果读者对其取值感兴趣的话，可以查阅 Win32 SDK 帮助，上面有详细的有关该标识符取值的说明。WS_EX_TOPMOST 是我惟一使用过的一个值，该功能使窗口一直保持在上部。
- **lpClassName**——这是你所创建的窗口的基础类名——例如“WINCLASS1”。
- **lpWindowName**——这是包含窗口标题的空终止文本字符串——例如“我的第一个窗口”。
- **dwStyle**——这是一个说明窗口外观和行为的通用窗口标志——非常重要！表 2.10 列出了一些最常用的值。当然，可以任意组合使用这些值来得到希望的各种特征。
- **x, y**——这是该窗口左上角位置的像素坐标。如果你无所谓，可使用 CW_USEDEFAULT，这将由 Windows 来决定。
- **nWidth, nHeight**——这是以像素表示的窗口宽度和高度。如果你无所谓，可使用 CW_USEDEFAULT，这将由 Windows 来决定。
- **hWndParent**——假如父窗口存在，这是指向父窗口的句柄。如果没有父窗口，桌面就是父窗口。
- **hMenu**——这是指向附属于该窗口菜单的句柄。下一章中将详细介绍，现在将其赋值 NULL。
- **hInstance**——这是应用程序实例。这里从 WinMain() 中使用实例。
- **lpParam**——高级特征，设置为 NULL。

表 2.10 列出了各种窗口标志设置。

表 2.10 dwStyle 的通用类型值

类 型	所创建的内容
WS_POPUP	弹出式窗口
WS_OVERLAPPED	带有标题栏和边界的重叠式窗口，类似 WS_TILED 类型

续表

类 型	所创建的内容
WS_OVERLAPPEDWINDOW	具有 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_THICKFRAME、WS_MAXIMIZEBOX 和 WS_MINIMIZEBOX 样式的重叠式窗口
WS_VISIBLE	开始就可见的窗口
WS_SYSMENU	标题栏上有窗口菜单的窗口
WS_BORDER	有细线边界的窗口
WS_CAPTION	有标题栏的窗口（包括 WS_BORDER 样式）
WS_ICONIC	开始就最小化的窗口，类似 WS_MINIMIZE 样式
WS_MAXIMIZE	开始就最大化的窗口
WS_MAXIMIZEBOX	具有最大化按钮的窗口。不能和 WS_EX_CONTEXTHELP 样式合并。WS_SYSMENU 也必须指定
WS_MINIMIZE	开始就最小化的窗口，类似 WS_ICONIC 样式
WS_MINIMIZEBOX	具有最小化按钮的窗口。不能和 WS_EX_CONTEXTHELP 样式合并。WS_SYSMENU 也必须指定
WS_POPUPWINDOW	带有 WS_BORDER、WS_POPUP 和 WS_SYSMENU 类型的弹出式窗口
WS_SIZEBOX	一个窗口边界可以变化，和 WS_THICKFRAME 类型相同
WS_HSCROLL	带有水平滚动条的窗口
WS_VSCROLL	带有垂直滚动条的窗口

注意：用黑体显示的是经常使用的值。

下面是使用标准控件在 (0,0) 位置创建一个大小为 400X400 像素的、简单的重叠式窗口。

```

HWND hwnd;    // window handle

// create the window, bail if problem
if (!(hwnd = CreateWindowEx(NULL,    // extended style
    " WINCLASS",                      // class
    " Your Basic Window",            // title
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0,0                               // initial x,y
    400,400                           // initial width,height
    NULL,                             // handle to parent
    NULL,                             // handle to menu
    hinstance,                        // instance of this application
    NULL)))                            // extra creation parms
    return( 0 );

```

一旦创建了该窗口，它可能是可见或不可见的。但是，在这个例子中，我们增加了自动显示的类型标识符 `WS_VISIBLE`。如果没有添加该标识符，则调用下面的函数来人工显示该窗口：

```
// this shows the window
ShowWindow(hwnd, ncmashow);
```

记住 `WinMain()` 中的 `ncmdshow` 参数了吗？这就是使用它的方便之处。尽管我们使用 `WS_VISIBLE` 覆盖了 `ncmdshow` 参数，但还是应将其作为一个参数传递给 `ShowWindow()`。

下面让 Windows 更新窗口的内容，并且产生一个 `WM_PAINT` 信息，这通过调用函数 `UpdateWindow()` 来完成：

```
// this sends a WM_PAINT message to window and makes
// sure the contents are refreshed
UpdateWindow();
```

事件处理程序

我并不了解你的情况，但注意我现在正使你掌握 Windows 的核心。它有如一本神秘小说。请记住，我所说的程序就是当事件发生时 Windows 从主事件循环调用的回调函数。回顾一下图 2.6，刷新一下你对通用数据流的印象。

事件处理器由读者自己编写，它能够处理你所关心的所有事件。其余的工作就交给 Windows 处理。当然，请记住，你的应用程序所能处理的事件和消息越多，它的功能越强。

在编写程序之前，让我们讨论一下事件处理器的一些细节，即事件处理器能做什么，工作机理如何。首先，对于创建的任何一个 Windows 类，都有一个独立的事件处理器，我指的是 Windows' Procedure，从现在开始简称 `WinProc`。当收到用户或 Windows 发送的消息并放在主事件序列中时，`WinProc` 就接收到主事件循环发送的消息。这简直是一个智力绕口令，我换个方式来说明……

当用户和 Windows 运行任务时，你的窗口和/或其他应用程序窗口产生事件和消息。所有消息都进入一个队列，而你的窗口的消息发送到你的窗口专用队列中。然后主事件循环检索这些消息，并且将它们发送到你的窗口的 `WinProc` 中来处理。

这几乎有上百个可能的消息和变量，因此，我们就不全部分析了。值得庆幸的是，你只需处理很少的消息和变量，就可以启动并运行 Windows 应用程序。

简单地说，主事件循环将消息和事件反馈到 `WinProc`，`WinProc` 对它们进行处理。因此不仅你要关注 `WinProc`，主事件循环同样也要关心 `WinProc`。现在我们简要地了解一下 `WinProc`，现假定 `WinProc` 只接收消息。

现在来看一下 WinProc 的工作机制，让我们看一下它的原型：

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // window handle of sender
    UINT msg,           // the message id
    WPARAM wParam,      // further defines message
    LPARAM lParam);     // further defines message
```

当然，这仅仅是回调函数的原型。只要将函数地址作为一个函数指针传递给 winclass.lpfnWndProc，就可以调用该函数的任何信息，如下所示：

```
winclass.lpfnWndProc = WindowProc;
```

参数的含义是不言自明的：

- **hwnd**——这是一个 Windows 句柄，只有当你使用同一个 Windows 类打开多个窗口时它才用到。这种情况下，hwnd 是表明消息来自哪个窗口的惟一途径。图 2.7 表示了这种情况。

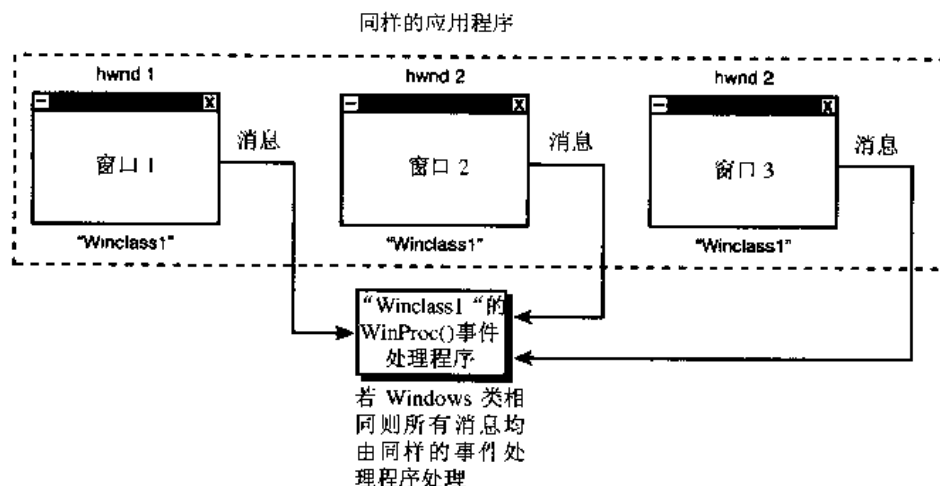


图 2.7 使用同一个类打开的多个窗口

- **msg**——这是一个实际的 WinProc 处理的消息标识符。这个标识符可以是众多主要消息中的一个。
- **Wparam** 和 **lparam**——进一步限定或细分发送到 msg 参数中的信息。

最后，我们感兴趣的是返回类型 **LRESULT** 和声明说明符 **CALLBACK**。这些关键字都是必需的，不能忽略它们。

因此大多数人所要做的就是使用 **switch()** 来处理 msg 所表示的消息，然后为每一种情况编写代码。在 msg 的基础上，你可以知道是否需要进一步求 wParam 和/或 lParam 的值。很酷吗？因此让我们看一下由 WinProc 传递过来的所有可能的消息，然后再看一下 WinProc 的工作机理。表 2.11 简要列出了一些基本的消息说明符。

表 2.11 消息说明符的简表

值	说 明
WM_ACTIVATE	当窗口被激活或者成为一个焦点时传递
WM_CLOSE	当窗口关闭时传递
WM_CREATE	当窗口第一次创建时传递
WM_DESTROY	当窗口可能要被破坏时传递
WM_MOVE	当窗口移动时传递
WM_MOUSEMOVE	当移动鼠标时传递
WM_KEYUP	当松开一个键时传递
WM_KEYDOWN	当按下一个键时传递
WM_TIMER	当发生定时程序事件时传递
WM_USER	允许传递消息
WM_PAINT	当一个窗口需重画时传递
WM_QUIT	当 Windows 应用程序最后结束时传递
WM_SIZE	当一个窗口改变大小时传递

要认真看表 2.11，了解所有消息的功能。在应用程序运行时将有一个或多个上述消息传递到 WinProc。消息说明符本身在 msg 中，而其他信息都存储在 wParam 和 lParam 中。因此参考在线 Win32 SDK 帮助来了解某个消息的参数所代表的意思是个不错的方法。

幸好我们现在只对下面三个消息感兴趣：

- WM_CREATE——当窗口第一次创建时传递该消息，以便你进行启动、初始化或资源配置工作。
- WM_PAINT——当一个窗口内容需重画时传递该消息。这可能有許多原因：用户移动窗口或改变其尺寸、弹出其他应用程序而遮挡了你的窗口等。
- WM_DESTROY——当窗口可能要被破坏时该消息传递到你的窗口。通常这是由于用户单击该窗口的关闭按钮，或者是从该窗口的系统菜单中关闭该窗口造成的。无论上述哪一种方式，都应当释放所有的资源，并且通过发送一个 WM_QUIT 消息来通知 Windows 完全终止应用程序。后面还将详细介绍。

OK！让我们看一看 WinProc 处理这些消息的整个过程。

```

LRESULT CALLBACK WindowProc(HWND hwnd,
                             UINT msg,
                             WPARAM wParam,
                             LPARAM lParam);

{
// this is the main message handler of the system
PAINTSTRUCT ps; // Used in WM_PAINT

```

```

HDC hdc;    // handle to a device context

// what is the message
switch(msg)
{
case WM_CREATE;
{
// do initialization stuff here

// return success
return ( 0 )
} break;

case WM_PAINT;
{
// simply validate the window
hdc = BeginPaint(hwnd, &ps);
// you would do all your painting here
EndPaint(hwnd, &ps);

// return success
return ( 0 )
} break;

case WM_DESTROY;
{
// Kill the application, this sends a WM_QUIT message
PostQuitMessage ( 0 );

// return success
return ( 0 )
} break;

default:break;

} // end switch

// process any message that we didn't take care of
return (DefWindowProc(hwnd, msg, wParam, lParam));

} // end WinProc

```

由上面可以看到，函数的大部分是由空白区构成——这真是件好事情。让我们就以 WM_CREATE 处理程序开始吧。该函数所作的一切就是 return(0)。这就是通知 Windows 由编程人员自己处理该函数，因此无需更多的操作。当然，也可以在 WM_CREATE 消息中进行全部的初始化工作，但那是你的事了。

下一个消息 WM_PAINT 非常重要。该消息在窗口需要重画时被发送。一般来说这表示

你应当进行重画工作。对于 DirectX 游戏来说，这并不是件什么大事，因为你将以 30 到 60 帧/秒的频率来重画屏幕。但是对于标准 Windows 应用程序来说，它就是件大事了。我将在后面章节中更详细地介绍 WM_PAINT，目前的功能就是通知 Windows，你要重画该窗口了，因此就停止发送 WM_PAINT 消息。

要完成该功能，你必须激活该窗口的客户区。有许多方法可以做到，但调用函数 `BeginPaint()` 和 `EndPaint()` 最简单。这一对调用将激活窗口，并使用原先存储在 Windows 类中的变量 `hbrBackground` 的背景刷来填充背景。下面是程序代码：

```
// begin painting
hdc = BeginPaint(hwnd, &ps);
// you would do all your painting here
EndPaint(hwnd, &ps);
```

下面要提醒几件事情。第一，请注意，每次调用的第一个参数是窗口句柄 `hwnd`。这是一个非常必要的参数，因为 `BeginPaint—EndPaint` 函数能够在任何应用程序窗口中绘制，因此该窗口句柄指示了要重画哪个窗口。第二个参数是包含必须重画矩形区域的 `PAINTSTRUCT` 结构的地址。下面是 `PAINTSTRUCT` 结构：

```
typedef struct tagPAINTSTRUCT
{
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

实际上不需要关心这个函数，当我们讨论图形设备接口时会再讨论这个函数。其中最重要的字段就是 `rcPaint`。图 2.8 表示了这个字段的内容。注意 Windows 一直尽可能地试图作最少的工作，因此当一个窗口内容破坏之后，Windows 至少会告诉你要恢复该内容并能够重画的最小的矩形。如果你对矩形结构感兴趣的话，会发现只有矩形的四个角是最重要的，如下所示：

```
typedef struct tagRECT
{
    LONG left;    // left x-edge of rect
    LONG top;     // top y-edge of rect
    LONG right;   // right x-edge of rect
    LONG bottom;  // bottom y-edge of rect
} RECT;
```

调用 `BeginPaint()` 函数应注意的最后一件事情是，它返回一个指向图形环境或 `hdc` 的句

柄:

```
HDC hdc;           // handle to graphics context
Hdc = BeginPaint(hwnd, &ps);
```

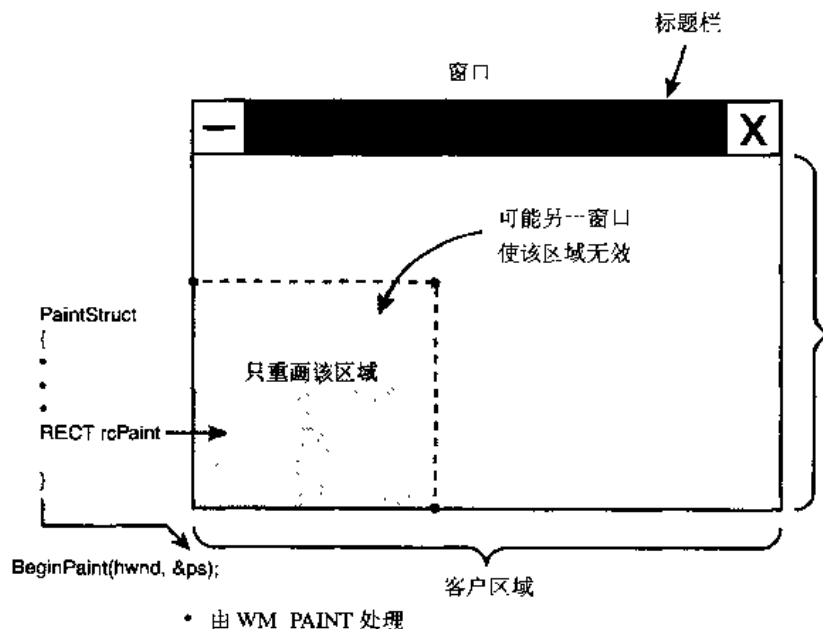


图 2.8 仅重新绘制无效区

图形环境就是描述视频系统和正在绘制表面的数据结构。奇妙的是，如果你需要绘制图形的话，只要获得一个指向图形环境的句柄即可。这便是关于 WM_PAINT 消息的内容。

WM_DESTROY 消息实际上非常有意思。WM_DESTROY 在用户关闭窗口时被发送。当然仅仅是关闭窗口，而不是关闭应用程序。应用程序继续运行，但是没有窗口。对此要进行一些处理。大多数情况下，当用户关闭主要窗口时，也就意味着要关闭该应用程序。因此，你必须通过发送一个消息来通知系统。该消息就是 WM_QUIT。因为该消息经常使用，所以有一个函数 PostQuitMessage() 来替你完成发送工作。

在 Wm_DESTROY 处理程序中你所要做的就是清除一切，然后调用 PostQuitMessage(0) 通知 Windows 终止应用程序。接着将 WM_QUIT 置于消息队列，这样在某一个时候终止主事件循环。

在我们所分析的 WinProc 句柄中还有许多细节应当了解。首先，你肯定注意到了每个处理程序体后面的 return(0)。它有两个目的：退出 WinProc 以及通知 Windows 你已处理的信息。第二个重要的细节是默认消息处理程序 DefaultWindowProc()。该函数是一个传递 Windows 默认处理消息的传递函数。因此，如果不处理该消息的话，可通过如下所示的调用来结束你的所有事件处理函数：

```
// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));
```

我认为代码本身过多并且过于麻烦。然而，一旦你有了一个基本 Windows 应用程序架构的话，你只要将它复制并在其中添加你自己的代码就行了。正如我所说的那样，我的主要目标是帮助你创建一个可以使用的类 DOS32 的游戏操作台，并且几乎忘记了任何正在运行的 Windows 工作。让我们转到下一部分——主事件循环。

主事件循环

最难的一部分终于结束了。我正要脱口而出：主事件循环太简单了。下面讨论一下：

```
// enter main event loop
while(GetMessage(&msg, NULL, 0, 0))
{
    // translate any accelerator keys
    TranslateMessage(&msg);

    // send the message to the window proc
    DispatchMessage(&msg);
} // end while
```

这是什么？OK！让我们来研讨一下。只要 GetMessage() 返回一个非零值，主程序 while() 就开始执行。GetMessage() 是主事件循环的关键代码，其惟一的用途就是从事件队列中获得消息，并进行处理。你会注意到 GetMessage() 有四个参数。第一个参数对我们非常重要，而其余的参数都可以设置为 NULL 或 0。下面列出其原型，以供参考：

```
BOOL GetMessage(
    LPMSG lpMsg,           // address of structure with message
    HWND hwnd,             // handle of window
    UINT wMsgFilterMin,    // first message
    UINT wMsgFilterMax);   // last message
```

msg 参数是 Windows 放置下一个消息的存储器。但是和 WinProc() 的 msg 参数不同的是，该 msg 是一个复杂的数据结构，而不仅仅是一个整数。当一个消息传递到 WinProc 时，它就被处理并分解为各个组元。MSG 的结构如下所示：

```
typedef struct tagMSG
{
    HWND hwnd;             // window where message occurred
    UINT message;          // message id itself
    WPARAM wParam;         // sub qualifies message
    LPARAM lParam;         // sub qualifies message
    DWORD time;            // time if message event
    POINT pt;              // position of mouse
} MSG;
```



```
// DEFINES ////////////////////////////////////////

// defines for windows
#define WINDOW_CLASS_NAME "WINCLASS1"

// GLOBALS ////////////////////////////////////////

// FUNCTIONS ////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
                             UINT msg,
                             WPARAM wparam,
                             LPARAM lparam)
{
    // this is the main message handler of the system
    PAINTSTRUCT ps;    // used in WM_PAINT
    HDC      hdc;      // handle to a device context

    // what is the message
    switch(msg)
    {
        case WM_CREATE:
        {
            // do initialization stuff here

            // return success
            return(0);
        } break;

        case WM_PAINT:
        {
            // simply validate the window
            hdc = BeginPaint(hwnd,&ps);
            // you would do all your painting here
            EndPaint(hwnd,&ps);

            // return success
            return(0);
        } break;

        case WM_DESTROY:
        {
            // kill the application, this sends a WM_QUIT message
            PostQuitMessage(0);

            // return success
            return(0);
        } break;
    }
}
```



```

        default: break;

    } // end switch

    // process any messages that we didn't take care of
    return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN //////////////////////////////////////
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    WNDCLASSEX winclass; // this will hold the class we create
    HWND        hwnd;    // generic window handle
    MSG         msg;      // generic message

    // first fill in the window class structure
    winclass.cbSize = sizeof(WNDCLASSEX);
    winclass.style = CS_DBLCLKS | CS_OWNDC |
                    CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra = 0;
    winclass.cbWndExtra = 0;
    winclass.hInstance = hinstance;
    winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;
    winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    // register the window class
    if (!RegisterClassEx(&winclass))
        return(0);

    // create the window
    if (!(hwnd = CreateWindowEx(NULL, // extended style
                              WINDOW_CLASS_NAME, // class
                              "Your Basic Window", // title
                              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                              0,0, // initial x,y
                              400,400, // initial width, height
                              NULL, // handle to parent
                              NULL, // handle to menu

```

```

        hinstance, // instance of this application
        NULL)))    // extra creation parms
return(0);

// enter main event loop
while(GetMessage(&msg, NULL, 0, 0))
{
    // translate any accelerator keys
    TranslateMessage(&msg);

    // send the message to the window proc
    DispatchMessage(&msg);
} // end while

// return to Windows like this
return(msg.wParam);

} // end WinMain

////////////////////////////////////

```

要编译 DEMO2_3.CPP，只需创建一个 Win32 环境下的 .EXE 应用程序，并且将 DEMO2_3.CPP 添加到项目中即可。假如你喜欢的话，可以直接从 CD-ROM 上运行预先编译好的程序 DEMO2_3.EXE。图 2.10 显示了运行中的该程序。



图 2.10 运行中的 DEMO2_3.EXE

在进行下一部分内容之前，我还有事情要说。首先，如果你认真阅读了事件循环的话，会发现它看上去并不是个实时程序。也就是说，当程序在等待通过 GetMessage() 传递的消息的同时，主事件循环基本上是锁定的。这的确是真的：你必须以各种方式来避免这种现象，因为你需要连续地运行你的游戏处理过程，并且在 Windows 事件出现时处理这些事件。

产生一个实时事件循环

这种实时的无等候的事件循环很容易实现。你所需要的就是一种测试在消息序列中是否有消息的方法。如果有，你就处理它；否则，继续处理其他的游戏逻辑并重复进行。运行的测试函数是 `PeekMessage()`。其原型几乎和 `GetMessage()` 相同，如下所示：

```
BOOL PeekMessage(
    LPMSG lpMsg,           // pointer to structure for message
    HWND hWnd,            // handle to window
    UINT wMsgFilterMin,    // first message
    UINT wMsgFilterMax,    // last message
    UINT wRemoveMsg);      // removal flags
```

如果有可用消息的话返回值非零。

区别在于最后一个参数，它控制如何从消息序列中检索消息。对于 `wRemoveMsg`，有效的标志有：

- `PM_NOREMOVE`——`PeekMessage()` 处理之后，消息没有从序列中去除。
- `PM_REMOVE`——`PeekMessage()` 处理之后，消息已经从序列中去除。

如果将这两种情况考虑进去的话，你可以做出两个选择：如果有消息的话，就使用 `PeekMessage()` 和 `PM_NOREMOVE`，调用 `GetMessage()`；另一种选择是：使用 `PM_REMOVE`，如果有消息则使用 `PeekMessage()` 函数本身来检索消息。一般使用后一种情况。下面是核心逻辑的代码，我们在主事件循环中稍作改动以体现这一新技术：

```
while(TRUE)
{
    // test if there is a message in queue, if so get it
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // test if this a quit
        if (msg.message == WM_QUIT)
            break;
        // translate any accelerator keys
        TranslateMessage( &msg );

        // send the message to the window proc
        DispatchMessage( &msg );
    } // end if

    // main game processing goes here
    Game_Main();
} // end while
```

我已经将程序中的重要部分用黑体显示。黑体的第一部分内容是：

```
if (msg.message == WM_QUIT) break;
```

下面是如何测试从无限循环体 `while(true)` 中退出。请记住，当在 `WinProc` 中处理 `WM_DESTROY` 消息时，你的工作就是通过调用 `PostQuitMessage()` 函数来传递 `WM_QUIT` 消息。`WM_QUIT` 就在事件序列中慢慢地移动，你可以检测到它，所以可以跳出主循环。

用黑体显示的程序最后一部分指出调用主游戏程序代码循环的位置。但是请不要忘记，在运行一幅动画或游戏逻辑之后，调用 `Game_Main()` 或者调用任意程序必须返回。否则，Windows 主事件循环将不处理消息。

这种新型的实时结构的例子非常适合于游戏逻辑处理程序，请看源程序 `DEMO2_4.CPP` 以及 CD-ROM 上相关的 `DEMO2_4.EXE`。这种结构实际上是本书剩下部分的模型。

打开多个窗口

在完成本章内容之前，我想讨论一个你可能非常关心的更重要的话题——如何打开多个窗口。实际上，这是小事一桩，其实你已经知道如何打开多个窗口。你所需要做的就是多次调用函数 `CreateWindowEx()` 来创建这些窗口，事实也的确如此。但是，对此还有一些需要注意的问题。

首先，请记住当你创建一个窗口时，是建立在 Windows 类的基础之上的。在该类中定义了 `WinProc` 或者整个类的事件处理程序。这是非常重要的细节，因此应当注意。你可以使用同一个类来创建多个窗口，这些窗口的所有消息都要传递到同一个 `WinProc` 中，正如由 `WINCLASSEX` 结构中定义 `lpfnWndProc` 字段指向的事件处理程序一样。图 2.11 表示了这种情况下的消息流。

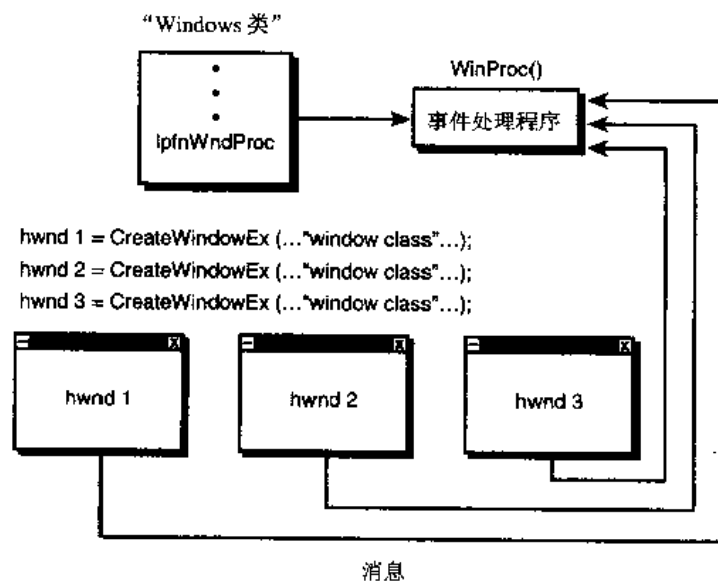


图 2.11 使用相同的 Windows 类打开多个窗口的消息流

这样可能达到你的愿望，也可能达不到。如果想应用不同的 **WinProc** 打开各个窗口的话，你必须创建多个 **Windows** 类，并且使用每一个类创建各个窗口。这样每一个类的窗口就指向各自的 **WinProc** 并向其传递消息。图 2.12 表示了这种情况。

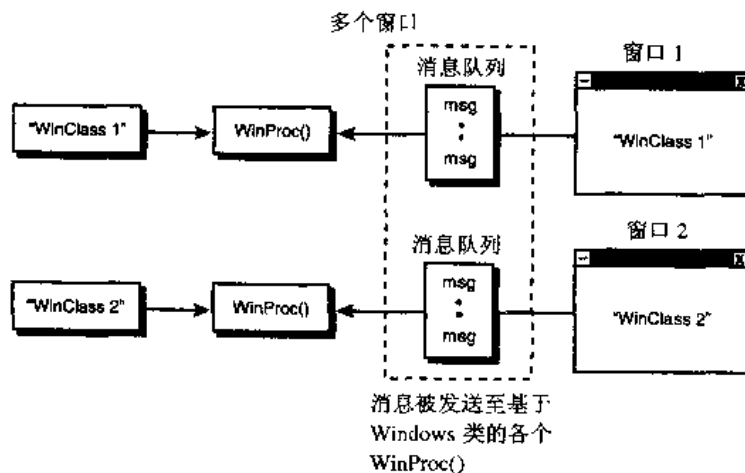


图 2.12 多个窗口的多个 Windows 类

了解了这些内容后，下面是基于同一个类来创建多个窗口的程序代码：

```
// create the first window
if (!(hwnd = CreateWindowEx(NULL,      // extended style
    WINDOW_CLASS_NAME,      // class
    "Window 1 Based on WINCLASS1",    // title
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0,0,      // initial x,y
    400,400  // initial width,height
    NULL,    // handle to parent
    NULL,    // handle to menu
    hinstance, // instance of this application
    NULL))) // extra creation parms
    return (0);

// create the second window
if (!(hwnd = CreateWindowEx(NULL,      // extended style
    WINDOW_CLASS_NAME,      // class
    "Window 2 Also Based on WINCLASS1", // title
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    100,100, // initial x,y
    400,400  // initial width,height
    NULL,    // handle to parent
    NULL,    // handle to menu
    hinstance, // instance of this application
    NULL))) // extra creation parms
    return (0);
```

当然，你可能希望用不同的变量而不是同一个变量来跟踪每一个窗口，就像 `hwnd` 中的例子那样，其实你已掌握了其要义，如一次打开两个窗口的例子。请看 `DEMO2_5.CPP` 以及 CD-ROM 上相关的可执行文件 `DEMO2_5.EXE`。当你运行 `.EXE` 执行文件时，应当可以看到如图 2.13 所示的情况。注意当你关闭任何一个窗口时，两个窗口将同时关闭，并且应用程序也终止。看一看是否能够找到每次只关闭一个窗口的方法。（提示：创建两个 `Windows` 类，直到两个窗口都关闭之后再传递 `WM_QUIT` 消息。）

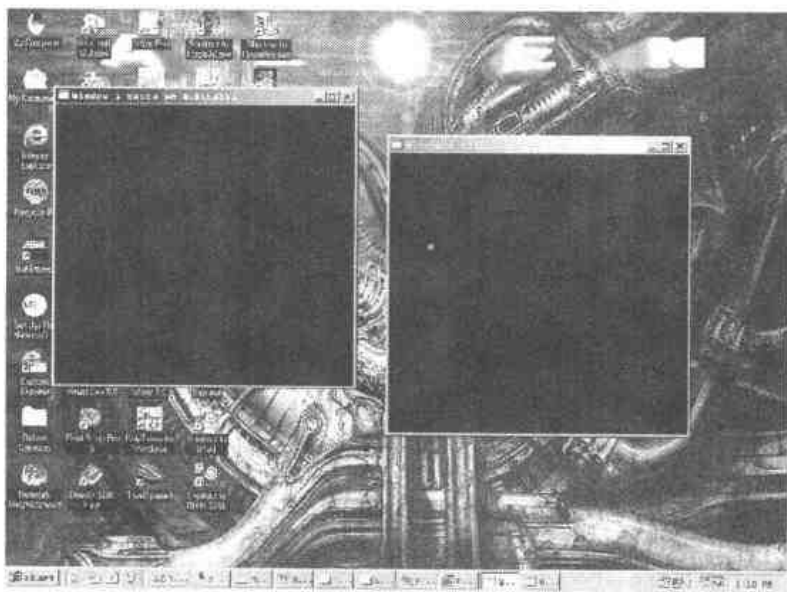


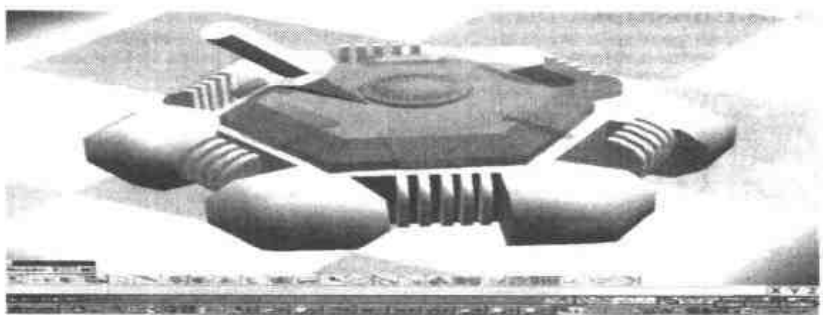
图 2.13 多窗口程序 `DEMO2_5.EXE`

总 结

尽管我并不了解你，但是我还是很为你激动！到目前为止，你已具备了 `Windows` 编程的基本知识并需要开始掌握更复杂的 `Windows` 编程知识。你已了解了 `Windows` 和多任务的结构，并且也知道了如何创建一个 `Windows` 类、注册类、创建窗口、编写事件循环和句柄等等内容。因此你已经打下了坚实的 `Windows` 编程基础。你完成了一项最杰出的任务。

下一章中，我们将了解更多的和 `Windows` 相关的内容，如：使用资源、创建菜单、使用对话框以及获取信息等。

3



高级 Windows 编程

在火箭专家看来，Windows 编程当然算不上什么大工程。但 Windows 编程很绝的地方在于，你不用了解太多细节，就可以完成很多工作。因此，我们在本章中将主要讨论开发一个完整的 Windows 应用程序所需要的一些最重要的内容。本章主要内容有：

- ➔ 使用资源（如图标、光标和声音）
- ➔ 菜单
- ➔ 基本的图形设备接口和视频系统
- ➔ 输入设备
- ➔ 传递消息

使用资源

Windows 创建者提出的一个主要设计目标就是，在一个 Windows 应用程序中除程序代码外还能储存更多的资源（甚至 Mac 程序也是如此）。他们认为一个程序的数据也能够驻留在该程序的.EXE 文件中。这是个不错的想法，因为：

- 一个同时含有代码和数据的.EXE 文件更容易分配。
- 如果没有外部数据文件的话，就不会丢失这些数据。
- 外部强制转移不会很容易地访问、任意删改、添加、和分配你的数据文件（例如，.BMP 文件、.WAV 文件等等）。

要满足这种数据库技术，Windows 程序支持该种功能，称之为资源。这只是与你的程序代码结合在一起的数据的一小部分，这部分数据在以后的运行过程中可被程序本身加载。

图 3.1 解释了这个概念。

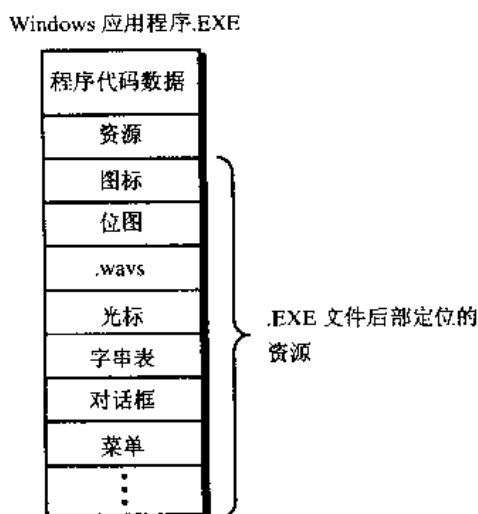


图 3.1 资源和 Windows 应用程序的关系

那我们讨论的是哪一种资源呢？实际上对于想编译进程序中的数据类型并没有什么限制，因为 Windows 程序支持用户定义的资源类型。但是应当注意几种预定义的类型：

- 图标——小位图图形，可以用于许多方面，例如单击该图形运行一个目录下的一个程序。图标使用.ICO 文件扩展名。
- 光标——一个表示鼠标指针的位图。Windows 允许以各种方式操作光标。例如，可以令光标在窗口之间移动时变换。光标使用.CUR 文件扩展名。
- 字符串——字符串资源对于作为一种资源来讲可能是最不明显的了。可以这样说：“我经常将字符串添加到我的程序中或者一个数据文件中。”我知道你的意思。然而，Windows 允许将一个字符串表作为一种资源放到你的程序中，并且通过标识符来访问它们。
- 声音——大部分 Windows 程序至少都可以通过.WAV 文件来使用声音。因此，.WAV 文件也是一种资源。
- 位图——这是标准的位图，可以是单色、4 位、8 位、16 位或 32 位格式的像素矩阵。在图形操作系统（如 Windows）中是非常常用的对象，因此也可以将位图作为一种资源。位图使用.BMP 文件扩展名。
- 对话框——对话框在 Windows 中也非常常用，设计者可以让对话框作为一种资源，而不是在外部装载的东西。好主意！因此，你可以在程序中创建对话框，也可以将它们设计为一个编辑器，作为一种资源来存储。
- 图元文件——图元文件相对高级。它们允许将一个图像操作作为一个序列记录在一个文件中，然后再回放它。

现在你已经了解了资源的定义以及存在形式，下一步就是如何将它们一起使用。好！

有一个资源编译器的程序，可以以一个扩展名为.RC 的 ASCII 文本资源文件输入。该文件是一个 C/English 文件——描述了编译到一个数据文件中的所有资源。该资源编译器装载所有的资源，以.RES 的扩展名形式将所有资源放置在一个大数据文件中。

这个.RES 文件包含了你在.RC 文件中定义的诸如图标、光标、位图、声音等所有资源的二进制数据。该.RES 文件和.CPP、.H、.LIB、.OBJ 等等文件一样都可以编译成一个.EXE 文件，这就已经足够了！图 3.2 显示了该过程的数据流程的可能性。

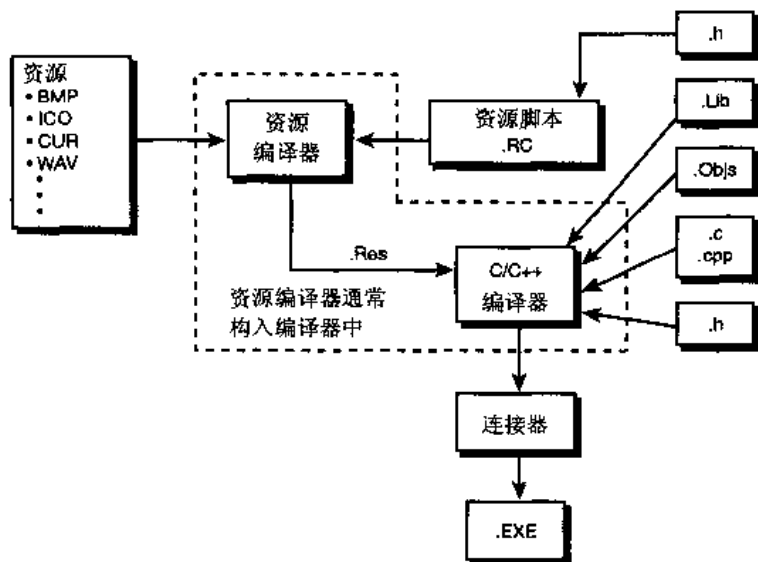


图 3.2 编译和连接过程中资源的数据流

集合资源

以前可以使用一个外部资源编译器，如 RC.EXE 将所有的资源编译到一起。但是现在可以使用编译器 IDE 来做这些工作。因此，如果在程序中添加一种资源的话，可以简单地通过 IDE 中的“文件”菜单中选择“新建”按钮（大多数情况下），然后选择想要添加的资源类型（后面将详细讨论）来添加资源。

让我们回顾一下如何处理资源：可以向程序中添加许多数据类型和对象，然后它们以资源的形式和实际程序代码一起驻留在.EXE 文件中（一般在文件的末尾某处）。在运行过程中，可以访问这个资源数据库，并且可以从程序本身（而不是作为一个单独的文件从磁盘中）装载资源数据。要创建该资源文件，必须有一个以 ASCII 文本形式的资源描述文件，名称为.RC。然后将该文件传递到编译器中（一起访问该资源），并且产生一个.RES 文件。然后将该.RES 文件和所有的其他程序对象连接到一起，创建一个最终的.EXE 文件。就这么简单！好，这样我就变成了一个亿万富翁了。

记住上述所有内容，下面就让我们讨论一下众多的资源对象，学会如何创建并装载到程序中。我就不再重复上面提到过的所有的资源，但是你应该能够指出任何其他的信息对

象。它们都以相同的方式运行，产生或采用一个数据类型、句柄或熬一个通宵不休息的精神插曲。

使用图标资源

使用资源只需要创建两个文件：一个是.RC 文件，如果想在.RC 文件中对符号标识符进行标注的话，可能还需要创建一个.H 文件。在下面内容中将详细讨论。当然，最后还需要产生一个.RES 文件，但是我们让 IDE 编译器来做这个工作。

作为一个创建图标资源的例子，让我们看一下如何改变任务栏上的应用程序使用的图标以及窗口本身的系统菜单的图标。如果你还记得的话，我们在 Windows 类的创建过程中使用下面代码设置过这些图标：

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

这两行代码为这些普通图标和小版本的图标加载默认的应用图标程序。实际上可以通过使用已经编译到一个资源文件中的图标，将所需要的图标装载到这些存取窗口中。

首先需要有一个图标，我已经创建了一个很酷的图标用于本书中所有的应用程序。该图标名为 T3DX.ICO，如图 3.3 所示。我使用了 VC++5.0 的图形编辑器（如图 3.4 所示）创建了该图标。当然你可以使用任何你想使用的程序（只要该程序支持该种输出类型）来创建图标、光标和位图等。



图 3.3 T3DX.ICO 图标位图

T3DX.ICO 是 32 像素×32 像素，16 位色彩。图标可以安排在从 16×16 到 64×64 的区域内，最高可以达到 256 色彩。但是大多数图标都是 32×32 以及 16 位色彩的，我们也采用这种格式。

一旦你有了感兴趣的、需要放入一个资源文件中的图标，就需要创建一个资源文件来放置该图标。为了简单起见，需要人工编写。（请记住，IDE 编译器完全可以做这些工作，但是那样的话，你就什么也学不到了，不是吗？）

.RC 文件包含所有资源的定义，也就是说在程序中使用多种资源。

注意



在编写任何代码之前，我想指出关于资源的非常重要的一点。Windows 可以使用 ASCII 文本字符串或者是整数标识符来引用资源。在大多数情况下，你可以在.RC 文件中同时使用这两种方式，但是应当注意一些资源只允许使用其中的一种。无论是哪种情况，资源必须以稍微不同的方式来加载，并且如果涉及到标识符的话，在你的工程中必须包含一个另外的符号前后对照的.H 文件。

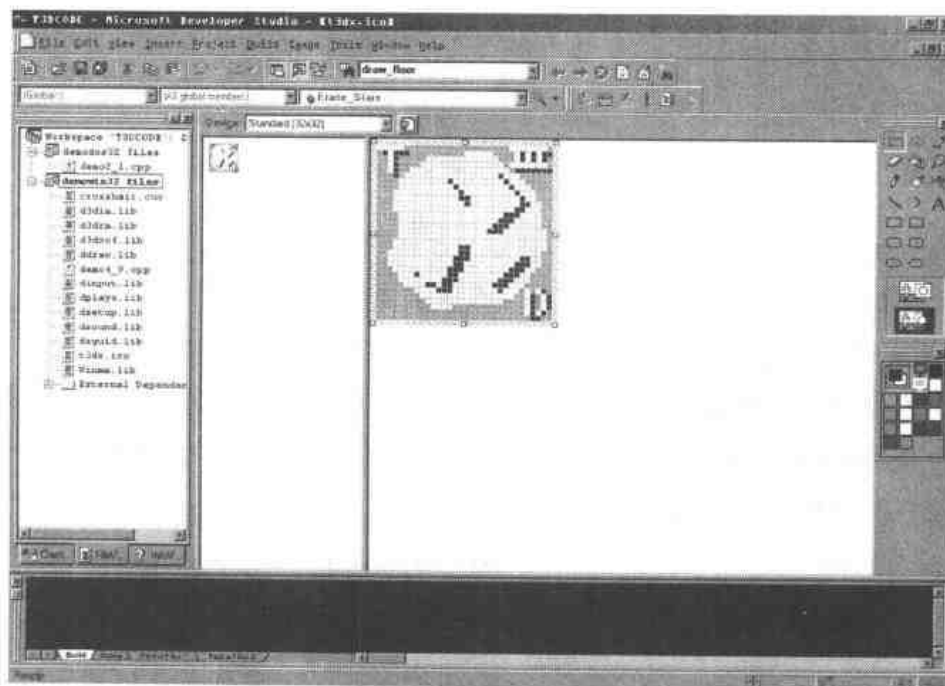


图 3.4 VC++5.0 图形编辑器

下面是如何在.RC 脚本文件中定义一个 ICON 资源:

方法 I——使用字符串名

```
icon_name ICON FILENAME.ICO
```

例如:

```
windowicon ICON star.ico
```

```
MyCoolIcon ICON cool.ico
```

或者是

方法 II——通过整型标识符

```
icon_id ICON FILENAME.ICO
```

例如:

```
windowicon ICON star.ico
```

```
124 ICON ship.ico
```

这是令人混乱的地方: 注意方法 I 中根本没有任何注解。这就是个问题, 容易给你带来麻烦, 因此要仔细听。应当能注意到: ICON 定义的每一种方法的第一个例子看上去完全一样。但是, 一个理解为“windowicon”, 而另一个是一个符号 windowicon。这样就导致在.RC 文件中必须包含一个附加文件, 来定义符号常量。当资源编译器解析下面代码时,

```
windowicon ICON star.ico
```

资源编译器首先看一下已经在 `include` 头文件定义的符号。如果该符号存在，资源编译器就通过整型标识符来引用该符号所指向的资源。否则，资源编译器就假定它是个字符串，通过字符串“`windowicon`”来引用 `ICON`。

因此，如果想在 `.RC` 资源脚本中定义符号 `ICON` 的话，也需要一个 `.H` 文件来解析该符号索引。要想在 `.RC` 脚本中包含 `.H` 文件，应当使用标准 `C/C++ #include` 描述符。

例如，假定你想在 `.RC` 文件 `RESOURCES.RC` 中定义三个符号图标，同时也需要一个 `.h` 文件 `RESOURCES.H`。下面是每个文件的内容：

RESOURCES.H 的内容：

```
#define ID_ICON1 100 // these numbers are arbitrary
#define ID_ICON2 101
#define ID_ICON3 102
```

RESOURCES.RC 的内容：

```
#include "RESOURCES.H"
// here are the icon defines, note the use of C++ comments
ID_ICON1 ICON star.ico
ID_ICON2 ICON ball.ico
ID_ICON3 ICON cross.ico
```

就是这样。然后可以将 `RESOURCES.RC` 添加到你的工程中，确认应用程序文件中有 `#include RESOURCES.H`，然后你就大功告成了。当然，`.ICO` 文件必须放在工程的工作目录下，以便于资源编译器能够找到它们。

如果没有为图标定义（`#define`）符号，也没有包含一个 `.H` 文件，资源编译器将只能假定符号 `ID_ICON1`、`ID_ICON2` 和 `ID_ICON3` 是字符串。然后就是如何在程序“`ID_ICON1`”、“`ID_ICON2`”和“`ID_ICON3`”中引用它们。

我已经完全打乱了所论及问题的时间/空间连续性，让我们回顾一下想做的工作——仅仅是加载一个简单的图标。

要想通过字符串名来装载一个图标，按下面步骤进行：

在 `.RC` 文件中：

```
your_icon_name ICON filename.ico
```

在程序代码中：

```
// Notice the use of hinstance instead of NULL.
```

```
Winclass.hIcon = LoadIcon(hinstance, "your_icon_name");
Winclass.hIconSm = LoadIcon(hinstance, "your_icon_name");
```

要通过符号参考来装载，可以如前面例子 `#include` 那些包含符号参考的头文件。
在.H 文件中：

```
#define ID_ICON1 100 // these numbers are arbitrary
#define ID_ICON2 101
#define ID_ICON3 102
```

在.RC 文件中：

```
// here are the icon defines, note the use of C++ comments
ID_ICON1 ICON star.ico
ID_ICON2 ICON ball.ico
ID_ICON3 ICON cross.ico
```

程序代码可以像下面一样：

```
// Notice the use of hinstance instead of NULL.
// use the MAKEINTRESOURCE macro to reference
// symbolic constant resource properly
Winclass.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(ID_ICON1));
Winclass.hIconSm = LoadIcon(hinstance, MAKEINTRESOURCE(ID_ICON1));
```

注意宏 `MAKEINTRESOURCE()` 的使用。该宏将整数转换为一个字符串指针，但是不必担心该操作——只在使用已经 `#define` 的符号常数时应用它。

使用光标资源

光标资源几乎和图标资源相同。光标文件是一个小位图，扩展名为 `.CUR`，可以在大多数 IDE 编译器中创建，或者使用单独的图像处理程序来创建。光标通常是 32×32 以及 16 位色彩的，最高可达 64×64 以及 256 色彩，甚至可以采用动画。

假定已经使用 IDE 或一个单独的绘图程序创建了一个光标文件，将它们添加到.RC 文件中以及通过程序来访问它们的步骤和图标的情况相似。要定义一个光标，使用.RC 文件中的光标关键字。

方法 I——通过字符串名

```
cursor_name CURSOR FILENAME.CUR
```

例如:

```
windowcursor CURSOR crosshair.cur
MyCoolCursor CURSOR greenarrow.cur
```

或者是

方法 II——通过整型标识符

```
cursor_id CURSOR FILENAME.CUR
```

例如:

```
windowcursor CURSOR bluearrow.cur
292 CURSOR redcross.cur
```

当然, 如果想使用符号标识符, 必须创建一个.H 文件以及符号的定义。

RESOURCES.H 的内容:

```
#define ID_CURSOR_CROSSHAIR 200 // these numbers are arbitrary
#define ID_CURSOR_GREENARROW 201
```

RESOURCES.RC 的内容:

```
#include "RESOURCES.H"
// here are the icon defines, note the use of C++ comments
ID_CURSOR_CROSSHAIR CURSOR crosshair.cur
ID_CURSOR_GREENARROW CURSOR greenarrow.cur
```

没有任何原因说明一个资源数据文件为什么不能存在于其他目录中。例如 greenarrow.cur 可能存在于一个 CURSOR 目录的根目录下, 像下面一样:

```
ID_CURSOR_GREENARROW CURSOR C:\CURSOR\greenarrow.cur
```

技巧



我已经为本章创建了一些光标.cur 文件。使用你的 IDE 来浏览一下这些光标文件, 或者仅打开该目录, Windows 将通过其文件名显示每个光标文件的位图。

现在已经了解了如何向一个.RC 文件中添加一个光标资源, 下面是按照字符串名从应用程序中加载该资源的程序代码。

在.RC 文件中:

```
CrossHair CURSOR crosshair.cur
```

在程序代码中:

```
// 注意使用 hinstance 而不是 NULL
Winclass.hCursor = LoadCursor(hinstance, "crossHair");
```

要通过.H 文件中定义的符号标识符来装载光标, 具体步骤如下。

在.H 文件中:

```
#define ID_CROSSHAIR 200
```

在.RC 文件中:

```
ID_CROSSHAIR CURSOR crosshair.cur
```

在程序代码中:

```
// 注意使用 hinstance 而不是 NULL
Winclass.hCursor = LoadCursor(hinstance, MAKEINTRESOURCE(ID_CROSSHAIR));
```

现在已经是第二次使用宏 MAKEINTRESOURCE()来将符号整型 ID 转换为 Windows 系统使用的格式。

好, 有一个细节可能还没有引起你的注意。到现在为止只遇到了 Windows 类图标和光标。但是在窗口之间操作窗口图标和光标可能吗? 例如, 如果想创建二个窗口, 并且使光标在两个窗口之间不同。要想做到这一点的话, 应当使用 SetCursor()函数:

```
HCURSOR SetCursor(HCURSOR hCursor);
```

其中, hCursor 是一个由 LoadCursor()检索的光标句柄。使用该技术惟一的问题就是 SetCursor()函数不太灵便, 因此应用程序中必须在鼠标从一个窗口移动到另一个窗口的同时跟踪和改变光标。下面是一个设置光标的例子:

```
// load the cursor somewhere maybe in the WM_CREATE
HCURSOR hcrosshair = LoadCursor(hinstance, "CrossHair");

// later in program code to change the cursor...
SetCursor(hcrosshair);
```

CD-ROM 上 DEMO3_1.CPP 给出了一个设置窗口图标和鼠标光标的例子。下面清单给出了加载新图标和光标的重要的代码部分的摘录。

```

/ include resources
#include "DEMO3_1RES.H"

.
.

// changes to the window class definition
winclass.hIcon =
    LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));
winclass.hCursor=
    LoadCursor(hinstance, MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.hIconSm = LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));

```

并且，程序使用了资源脚本程序 DEMO3_1.RC 和资源头文件 DEMO3_1RES.H。
DEMO3_1RES.H 的内容：

```

#define ICON_T3DX          100
#define CURSOR_CROSSHAIR  200

```

DEMO3_1.RC 的内容：

```

#include "DEMO3_1RES.H"

// note that this file has different types of resources
ICON_T3DX  ICON t3dx.ico
CURSOR_CROSSHAIR CURSOR crosshair.cur

```

要自己建立应用程序，你需要下面文件：

DEMO3_1.CPP——C/C++主文件
DEMO3_1RES.H——定义符号的头文件
DEMO3_1.RC——资源脚本程序
T3DX.ICO——图标的位图数据
CROSSHAIR.CUR——光标的位图数据

所有这些文件都应当放在同一个目录下作为一个工程。否则编译器和连接器很难找到它们。一旦创建并运行了该程序，或者使用预编译程序 DEMO3_1.EXE，就应当看到如图 3.5 所示的图像，非常酷，是吧！



图 3.5 关于常用图标和光标的 DEMO3_1.EXE 的输出结果

作为一个试验，尝试用 IDE 打开 DEMO3_1.RC 文件。图 3.6 表示了我使用 VC++5.0 所作的内容。但是，使用专用编译器应当得到不同的结果，因此如果结果不相同也不必大惊小怪。OK！在进入下面内容之前讨论一下 IDE。如前所述，可以使用 IDE 来创建.RC 和.H 文件，但是，应当阅读一下 IDE 的手册。



图 3.6 在 VC++5.0 环境下打开 DEMO3_1.RC 源程序的结果

但是装载一个人工编写的.RC 文件还有一个问题，如果使用你的 IDE 保存该文件的话，毫无疑问 Windows 编译器认为在.RC 文件中造成了无数的注释、宏、定义（#define）以及其他垃圾数据。实际上该问题的本质是如果想编辑人工编写的.RC 文件的话，应当将.RC 文

件作为文本来加载来进行编辑。这样编译器就不会将它作为一个.RC 文件来装载，但是应当注意只能是无格式的 ASCII 文本。

创建字符串表格资源

如引言中所提到的，Windows 支持字符串资源。和其他资源不同的是，只能使用一个字符串表单来容纳所有的字符串。

并且字符串资源不允许通过字符串来定义。因此，在.RC 文件中定义的所有字符串表单必须和符号引用常量以及相关的解释该引用的.H 头文件同时使用。

我依然不能确认我使用字符串资源的感受。使用字符串资源实际上和使用头文件相同，并且两种情况下——字符串或无格式头文件——都必须重新编译。因此，我认为没有使用它们的必要。但如果你确实不嫌麻烦，可以将字符串资源放入.DLL 文件中，主程序就不必要再重新编译它们。但是，我是一个技术人员，不是思想家，谁去关心这些问题？

要在.RC 文件中创建一个字符串表单，必须使用下面语法：

```
STRINGTABLE
{
    ID_STRING1, "string 1"
    ID_STRING2, "string 2"
    .
    .
}
```

当然，符号常量可以是任何东西，就像注释中的字符串一样。只有一条原则：每行字符串不能超过 255 个字符（包括常量本身在内）。

下面是一个包含可能在游戏菜单中使用的字符串表单的.H 和.RC 文件的例子。.H 文件包含有：

```
// 常数值取决于你
#define ID_STRING_START_GAME 16
#define ID_STRING_LOAD_GAME 17
#define ID_STRING_SAVE_GAME 18
#define ID_STRING_OPTIONS 19
#define ID_STRING_EXIT 20
```

.RC 文件中含有：

```
// note the stringtable does not have a name since
// only one stringtable is allowed per .RC file
STRINGTABLE
{
    ID_STRING_START_GAME, "Kill Some Aliens"
    ID_STRING_LOAD_GAME "Download Logs"
```

```

ID_STRING_SAVE_GAME    "Upload Data"
ID_STRING_OPTIONS      "Tweak The Settings"
ID_STRING_EXIT         "Let's Bail!"
}

```

技巧



你几乎可以将所有的想添加的内容都放到字符串表中，包括 `printf()` 命令格式如 `%d`、`%s` 等等。不能使用换行符 `"\n"`，但是可以使用八进制换码顺序如 `\015` 等等。

一旦创建了含有字符串资源的源文件，应当使用 `LoadString()` 函数来装载某一个字符串，下面是 `LoadString()` 函数的原型：

```

int LoadString(HINSTANCE hInstance, // handle of module with string resource
               UINT uID,             // resource identifier
               LPTSTR lpBuffer,      // address of buffer for resource
               int nBufferMax);      // size of buffer

```

`LoadString()` 返回一个所读字符的数量，或者在调用不成功时返回 0。下面是如何在游戏运行过程中调用和保存游戏字符串的函数：

```

// create some storage space
char load_string[80], // used to hold load game string
    save_string[80],  // used to hold save game string

// load in the first string and check for error
if (!LoadString(hinstance, ID_STRING_LOAD_GAME, load_string, 80))
{
    // there's an error
} //end if

// load in the second string and check for error
if (!LoadString(hinstance, ID_STRING_SAVE_GAME, save_string, 80))
{
    // there's an error
} //end if

// use the strings now

```

和前面一样，`hinstance` 是应用程序在 `WinMain()` 中传递的实例。

上面已经包含了所有的字符串资源的用法。如果你能为它们找到更好的用处，请给我发一个 email: ceo@games3d.com。

使用声音 .WAV 资源

到现在为止，或许你已经能够方便地使用资源脚本程序，又或许你已经非常厌烦以至于打算攻击我的 Web 页，来对我进行破坏了。请记住，那是 Microsoft 公司

(<http://www.microsoft.com>) 开发的内容。我仅仅是了解它而已。

好吧，我已经向你声明了不经常使用的否认声明，让我们继续下面的装载声音资源的内容！

大多数游戏都是用两种声音类型中的一种：

- 数字化 .WAV 文件
- MIDI .MID 文件

据我所知，Windows 的标准资源仅支持.WAV 文件，因此我就只分析如何创建.WAV 资源。当然如果不支持.MID 资源的话，你可能只能创建用户定义的资源类型。我们现在不讨论这个主题，但别处会加以讨论。

所需要的第一个事情就是一个.WAV 文件，它只是一个含有大量的 8 位或 16 位的一定频率脉冲的数字式数据波形。对于游戏声音效果来讲，典型的脉冲频率为 11MHz、22 MHz 和 44 MHz（CD 级品质）。该内容仍然和你无关，但我希望对该内容概述一下。在我们学习 DirectSound 时，将会学习有关数字采样理论和.WAV 文件的全部内容。现在，只讨论一下样本大小和频率。

我们假设你的磁盘上已经有了.WAV 文件，并且希望将该文件添加到一个源文件中，能够装载并且按照一定的程序播放。Let's go on! .WAV 文件的源类型就是 WAVE——这是令人惊奇的事。要将该文件添加到.RC 文件中，应当使用下面语法：

方法 I—通过字符串名

```
wave_name WAVE FILENAME.WAV
```

例如：

```
BigExplosion WAVE expl1.wavr
FireWeapons WAVE fire.wav
```

方法 II—通过整型标识符

```
ID_WAVE CURSOR FILENAME.CUR
```

例如：

```
DEATH_SOUND_ID WAVE die.wav
20                WAVE intro.wav
```

当然，该符号常量应当在一个.H 文件中的某处进行定义，这部分内容我们已经有所了解。

对于这一点，我们可能碰上了一个小障碍：WAVE 资源要比光标、图表和字符串表单要复杂一点。所以问题是装载声音资源的程序要比装载其他资源要复杂一些，因此我们现在就不介绍在一个实际游戏中装载.WAV 资源的方式，在后面再详细介绍。现在介绍一下使

用 `PlaySound()` 函数装载和播放 WAV 文件的技巧。下面是 `PlaySound()` 函数的原型:

```
BOOL PlaySound(LPCSTR pszSound,    // string of sound to play
               HMODULE hmod,       // instance of application
               DWORD fdwSound);    // flags parameter
```

和 `LoadString()` 不同的是, `PlaySound()` 稍微有点复杂, 因此我们要深入了解一下每一个参数:

- **PszSound**——该参数或者是资源文件中声音资源的字符串名, 或者是磁盘上的文件名。并且可以使用 `MAKEINTRESOURCE()` 并且应用使用符号常量定义的一个 `WAVE`。
- **Hmod**——装载该资源的应用程序实例。它只是一个应用程序实例。
- **FdwSound**——这是个关键参数。该参数控制声音如何装载和播放。表 3.1 列出了 `FdwSound` 的一些最有用的值。

表 3.1 `PlaySound()` 函数的 `FdwSound` 参数的值

值	描 述
<code>SND_FILENAME</code>	该 <code>PszSound</code> 参数是文件名
<code>SND_RESOURCE</code>	该 <code>PszSound</code> 参数是一个资源标识符; <code>hmod</code> 必须辨别包含该资源的实例
<code>SND_MEMORY</code>	装载到 <code>RAM</code> 中的声音事件文件。 <code>FdwSound</code> 参数指定的该参数必须指向内存中的声音文件的图像
<code>SND_SYNC</code>	声音事件的同步反馈。声音事件播放完毕后, <code>PlaySound()</code> 返回。
<code>SND_ASYNC</code>	声音事件的异步播放, 开始播放声音后, <code>PlaySound()</code> 立即返回。要终止异步播放的波形声音, 调用 <code>PlaySound()</code> , 并且 <code>PszSound</code> 设为 <code>NULL</code>
<code>SND_LOOP</code>	声音重复播放直到再次调用 <code>PlaySound()</code> , 并且 <code>PszSound</code> 设为 <code>NULL</code> , 并且需要指定 <code>SND_ASYNC</code> 标识符来制定一个异步声音事件
<code>SND_NODEFAULT</code>	不使用默认声音事件。如果没有发现声音文件, <code>PlaySound()</code> 返回静音而不播放默认声音
<code>SND_PURGE</code>	因为调用任务而终止声音。如果 <code>PszSound</code> 不为 <code>NULL</code> 的话, 将停止所有指定声音的事件。如果 <code>PszSound</code> 为 <code>NULL</code> 的话, 将停止所有代表调用任务的声音
<code>SND_NOSTOP</code>	指定声音事件将让位于另外一个已经播放的声音事件。如果一个声音由于产生该声音所需要的资源正在播放其他声音文件而不能播放当前声音, 该函数不播放所要求的声音, 而是立即返回错误
<code>SND_NOWAIT</code>	如果驱动器正忙, 不播放声音, 该函数就立即返回

要使用 `PlaySound()` 播放一个 WAVE 声音资源，一般需要下面四个步骤：

1. 创建 WAV 文件并存储在磁盘上。
2. 创建 RC 资源脚本程序以及相关的头文件。
3. 编译该资源和程序代码。
4. 使用 `MAKEINTRESOURCE()` 宏，通过 WAVE 资源名或者是通过 WAVE 资源标识符在程序中设定一个 `PlaySound()` 的调用。

让我们看几个例子。首先是有两种声音的常规 RC 文件：一个是字符串名的声音文件，另一个符号常量的声音文件，分别命名为 `RESOURCE.RC` 和 `RESOURCE.H`。该文件如下：

`RESOURCE.H` 文件包含：

```
#define SOUND_ID_ENERGIZE 1
```

`RESOURCE.RC` 文件包含：

```
# include " RESOURCE.H"

// first the string name definend sound resource
"Teleporter WAVE teleport.wav

// and now the symbolically defined sound
SOUND_ID_ENERGIZE WAVE energize.wav
```

在程序中，下面显示了如何以不同方式播放声音：

```
// to play the telport sound asynchronously
PlaySound("Teleporter", hinstance,
          SND_ASYNC | SND_RESOURCE);

// to play the telport sound asynchronously with looping
PlaySound("Teleporter", hinstance,
          SND_ASYNC | SND_LOOP | SND_RESOURCE);

// to play the energize sound asynchronously
PlaySound(MAKEINTRESOURCE(SOUND_ID_ENERGIZE), hinstance,
          SND_ASYNC | SND_RESOURCE);

// and if you simply wanted to play a sound off disk
// directly then you could do this
PlaySound("C:\\path\\filename.wav", hinstance,
          SND_ASYNC | SND_FILENAME);
```

要停止所有的声音，使用 `SND_PURGE` 标识符并将声音名设为 `NULL`，如下所示：

```
// stop all sounds
PlaySound(NULL, hinstance, SND_PURGE);
```

很明显，有许多标识符选项可以自由选择匹配。但是仍然没有任何控制或菜单，所以很难对演示应用程序产生影响。作为一个简单的使用声音资源的演示程序来讲，我已经创

建了 DEMO3_2.CPP，可以从磁盘上找到。下面就将该程序列出清单，但是 99% 的程序都是曾经使用过的标准的模板，而声音代码也和前面例子中的代码行基本相同。该演示程序是预编译的程序，可以运行 DEMO3_2.EXE 来浏览。

但是我还是想列出所使用的 .RC 文件和 .H 文件，分别是 DEMO3_2.RC 和 DEMO3_2RES.H 文件：

DEMO3_2RES.H 文件内容：

```
// defines for sound ids
#define SOUND_ID_CREATE 1
#define SOUND_ID_MUSIC 2

// defines for icons
#define ICON_T3DX 500

// defines for cursors
#define CURSOR_CROSSHAIR 600
```

DEMO3_2.RC 文件内容：

```
#include "DEMO3_2RES.H"

// the sound resources
SOUND_ID_CREATE WAVE create.wav
SOUND_ID_MUSIC WAVE techno.wav

// icon resources
ICON_T3DX ICON T3DX.ICO

// cursor resources
CURSOR_CROSSHAIR CURSOR CROSSHAIR.CUR
```

可以看到我在代码中也包含了图标和光标资源，从而使程序更具趣味性。

要编制 DEMO3_2.CPP，我采用了标准 Windows 演示程序，并且在两部分内容中添加了声音代码的调用：WM_CREATE 消息和 WM_DESTROY 消息。在 WM_CREATE 消息处，设置了两个声音效果，一个声音在说：“创建窗口 (Creating window)”，然后停止；另一个是以循环模式播放的一小段音乐，能够一直播放下去。在 WM_DESTROY 消息部分停止所有的声音。

注意

第一段声音，我使用 SND_SYNC 标志。使用该标志是因为 PlaySound() 一次只允许播放一个声音，而且在播放过程中，我不希望第二段声音终止第一段声音。

下面是从 DEMO3_2.CPP 文件中添加到 WM_CREATE 消息和 WM_DESTROY 消息中的代码：

```

case WM_CREATE:
{
    // do initialization stuff here

    // play the create sound once
    PlaySound(MAKEINTRESOURCE(SOUND_ID_CREATE),
              hinstance_app, SND_RESOURCE | SND_SYNC);

    // play the music in loop mode
    PlaySound(MAKEINTRESOURCE(SOUND_ID_MUSIC),
              hinstance_app, SND_RESOURCE | SND_ASYNC | SND_LOOP);

    // return success
    return(0);
} break;

case WM_DESTROY:
{
    // stop the sounds first
    PlaySound(NULL, hinstance_app, SND_PURGE);

    // kill the application, this sends a WM_QUIT message
    PostQuitMessage(0);

    // return success
    return(0);
} break;

```

从上面代码中，可以发现有一个变量 `hinstance_app`，用来作为 `PlaySound()`调用的应用程序实例句柄。这只是一个全局变量，用来保存 `WinMain()`中传递的实例。它的代码紧跟在 `WinMain()`中类定义的后面，如下所示：

```

.
.
// 在全局变量中保存实例
hinstance_app = hinstance;

// 注册该窗口类
if (!RegisterClassEx(&winclass))
    return(0);
.
.

```

要建立该应用程序，在工程中需要包含下列文件：

DEMO3_2.CPP——源文件主程序。

DEMO3_2.RES.H——包含所有符号的头文件。

DEMO3_2.RC——源文件脚本程序。

TECHNO.WAV——音乐片断，需要放在工作目录下。

CREATE.WAV——创建窗口声音，需要放在工作目录下。

WINMM.LIB——Windows 多媒体库扩展。该文件位于编译器的 LIB\目录下。应当将该文件添加到从该目录到输出的所有工程中。

MMSYSTEM.H——WINMM.LIB 的头文件，该文件已被包含在 DEMO3_2.CPP 和我的所有演示程序中。你所应了解的就是该文件应当位于你的编译器搜索路径中。它也是标准 Win32 头文件集中的一部分。

使用编辑器创建 .RC 文件

创建 Windows 应用程序的大多数编译器都需要一个相当大的开发环境，如 Windows 的 Visual Development Studio 等。每一个 IDE 都含有一个或多个使用即插即用技术的自动创建不同资源、资源脚本程序和相关头文件的工具。

使用这些工具的惟一问题是要学习这些工具！并且，使用 IDE 创建的 .RC 文件都是人工可读写的 ASCII 码文本，但是又有大量的添加的定义（#define）和宏，编译器能够添加它们，以便于自动完成并且简化常量的选择和 MFC 界面工作。

因为现在我是 Microsoft VC++5.0 用户，我简要说明一下关于使用 VC++5.0 资源处理支持的关键要素。首先，向工程中添加资源可以有两种方式：

方法 I——从主菜单中使用 File、New 选项，可以向工程中添加大量的资源。图 3.7 就是进行该操作后产生的对话框屏幕图。当添加图标、光标、位图等资源时，IDE 编译器自动生成 Image Editor（如前面图 3.4 所示）。这是一个原始的图像编辑单元，可以用来绘制光标和图标。如果要添加一个菜单资源（该部分内容将在下面部分讨论），就会出现菜单编辑器。

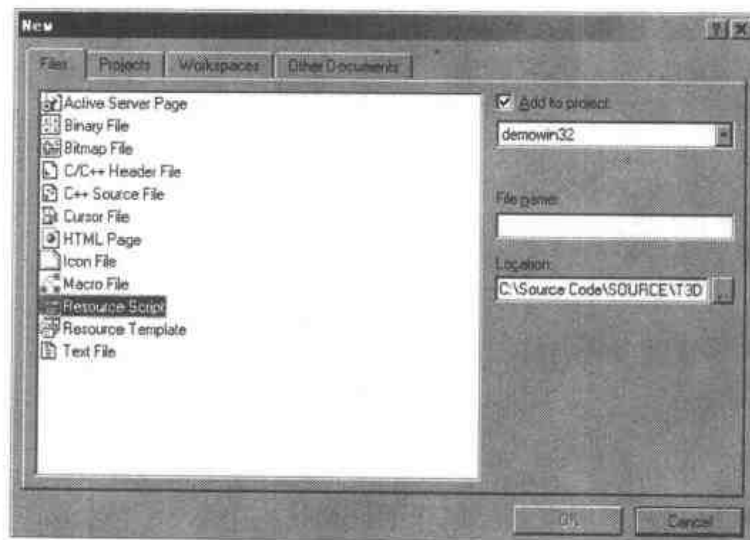


图 3.7 使用 VC++5.0 中的文件、新建来添加资源

方法 II——该方法更灵活一点，含有所有可能的资源类型，而方法 I 只支持其中的一部分。要向工程中添加任何一种类型的资源，可以使用主菜单中的 **Insert**、**Resource** 选项。图 3.8 表示了显示出的对话框。在这种情况下，你还需要进行一些维护（手工进行编辑）。无论何时想添加一个资源，都必须将它添加到资源脚本程序中——对吗？因此，如果你的工程中还没有一个资源脚本程序的话，编译器 IDE 将为你产生一个脚本程序，称之为 **SCRIPT*.RC**。另外，这两种方法最终都将生成一个名为 **RESOURCE.H** 的文件。该文件含有使用编辑器定义的和资源有关的资源符号、标识符值等。

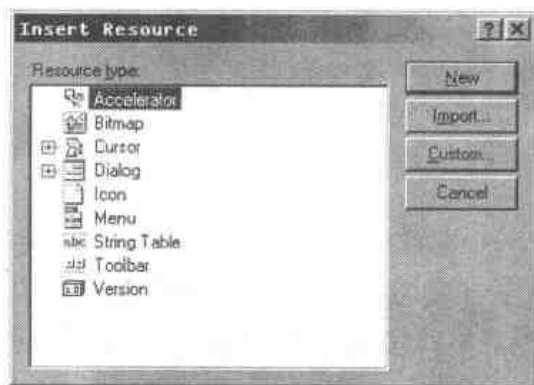


图 3.8 使用插入资源来向应用程序中添加资源

我当然希望深入探讨有关使用 IDE 进行资源编辑的内容，但由于它只是整章内容的一部分，而不是一整本书，所以请你自己阅读一下关于使用编译器进行文档编制的内容。本书中我们将不会使用更多的资源，因此上述讨论的信息已经足够了。下面让我们进行更复杂的资源类型——菜单的学习吧。

使用菜单编程

菜单是 Windows 程序中最酷的一个内容，可以说是人机交互式界面的关键所在（例如制造一个文字处理器）。了解如何创建和使用菜单是非常重要的，这是因为你可能想制作简单的工具来帮助创建游戏，或者是想有一个视窗基础的前端来作为游戏的开始。而这些工具毋庸置疑地要有大量的菜单（如果要制作一个 3D 工具的话，可能需要上百万个菜单）。请相信我！无论是哪种情况，都必须掌握如何创建、装载和响应菜单。

创建一个菜单

使用编译器的菜单编辑器可以创建一个完整的菜单以及相关的文件，但是现在我们人工编写它，因为我不知道你在使用哪种编译器，这样你也可以学会菜单说明书中的内容。但是当你正在编写一个实际的应用程序创建一个菜单时，大多数是应用 IDE 编辑器来创建菜单，因为人工输入编写菜单实在是太复杂了。它就像 HTML 代码——当启动 Web 后，使

用一个文本编辑器来制作一个主页并不是一件很复杂的事。但是，不使用任何工具来创建 Web 站点几乎是不可能的。

现在，就让我们开始制作菜单！实际上菜单和已经讨论过的其他资源完全一样。它们驻留在一个 .RC 资源脚本程序中，并且必须拥有一个 .H 文件来解决符号引用问题，符号引用是所有菜单的标识符（一个例外：菜单名必须是符号的，无名称字符串）。下面是在 .RC 文件中常见的一个菜单说明的基本语法：

```
MENU_NAME MENU DISCARDABLE
{ // you can use BEGIN instead of { if you wish

// menu definitions

} // you can use END instead of } if you wish
```

MENU_NAME 可以是一个名称字符串或这是一个符号，关键字 DISCARDABLE 是不完整的但又是必须的。看上去非常简单，当然，中间内容省略了，我会在后面讨论！

在编写定义菜单项和子菜单的代码之前，我们首先了解一些直观的相关术语。上述讨论可以参见图 3.9 中菜单，它有两个一级菜单：File 和 Help。File 菜单中包含四个菜单项：Open、Close、Save 和 Exit。帮助菜单中只有一个菜单项：About。因此说本菜单中含有一级菜单和菜单项。但是，这容易令人造成误解，因为菜单中还可能有菜单或者是层叠式菜单。我不准备创建层叠式菜单，但是层叠式菜单的原理很简单：使用一个菜单定义来定义一个菜单项本身，并且可以循环无休止的这样做下去。

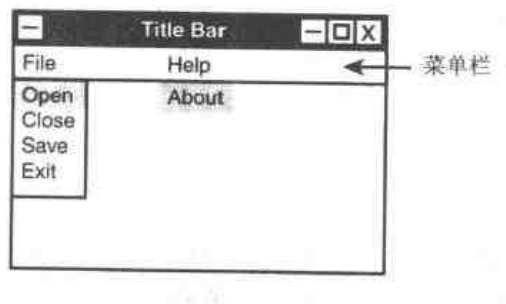


图 3.9 有两个子菜单的菜单栏

现在我们了解了相关术语，下面是执行图 3.9 所示的菜单：

```
MainMenu MENU DISCARDABLE
{
  POPUP "File"
  {
    MENUITEM "Open", MENU_FILE_ID_OPEN
    MENUITEM "Close", MENU_FILE_ID_CLOSE
    MENUITEM "Save", MENU_FILE_ID_SAVE
    MENUITEM "Exit", MENU_FILE_ID_EXIT
  }
  POPUP "Help"
  {
    MENUITEM "About", MENU_HELP_ID_ABOUT
  }
}
```

```

    } // end popup

    POPUP "Help"
    {
        MENUITEM "About", MENU_HELP_ABOUT
    } // end popup

} // end top level menu

```

我们来逐段分析该菜单定义。首先该菜单名为 **MainMenu**。对于这一点，我们并不知道它是一个名称字符串，还是一个标识符，但是因为我一般都大写所有常量的第一个字母，很清楚它是一个无格式字符串。这也正是我要做的内容。向下看，有两个一级菜单定义，都以关键字 **POPUP** 开头——这就是关键所在。**POPUP** 指出了—一个菜单可以使用下面 ASCII 名称和菜单项来定义。

ASCII 名必须跟在关键字 **POPUP** 后面，并且用括号来包起来。弹出式菜单定义必须放在 { } 或 **BEGIN END** 块中——喜欢使用哪种都可以。（使用 Pascal 语言的人应当高兴了。）

在该定义块中，后面是所有的菜单项。要定义一个菜单项，应当以下面语法使用关键字 **MENUITEM**：

```
MENUITEM " name ", MENU_ID
```

就是这样。当然在该例子中，没有发现任何符号，但是可以在 .H 文件中看到如下所示的格式：

```

// defines for the top level menu FILE
#define MENU_FILE_ID_OPEN    1000
#define MENU_FILE_ID_CLOSE   1001
#define MENU_FILE_ID_SAVE    1002
#define MENU_FILE_ID_EXIT    1003

// defines for the top level menu HELP
#define MENU_HELP_ABOUT      2000

```

提 示

注意标识符的值。我选择 1000 为第一个一级菜单的开始，然后每个菜单项加 1。下一个一级菜单在 1000 基础上再加 1000。由此每一个最高级级别的菜单间相差 1000，菜单中的每一个菜单项相差 1。这是一种良好的工作习惯，并且很少填充。

我没有定义 “**MainMenu**”，因为我希望通过字符串而不是标识符来访问该菜单。这并不是惟一的方式。例如，假如我在 .H 文件中将一行代码和其他符号放在一起，

```
#define MainMenu    100
```

资源编译器将自动假定我希望通过标识符来访问该菜单。我就必须使用

MAKEINTRESOURCE(MainMenu)或 MAKEINTRESOURCE(100)来访问该菜单资源。知道了吗？好，继续吧！

技巧



注意许多菜单项都有热键或快捷键，可以不必使用鼠标手动选择一级菜单或菜单项。可以使用&符号来达到目的。所要做的工作就是在 POPUP 菜单或 MENUITEM 字符串中将该符号&放在想标识热键或快捷键的符号前面。例如：

```
MENUITEM " E&xit", MENU_FILE_ID_EXIT
```

这样 X 就成为热键，

```
POPUP " &File"
```

通过 ALT+F 使得 F 成为一个快捷键。

现在已经了解了如何创建和定义一个菜单，让我们看一看如何将它装载到应用程序中以及如何连接到一个窗口。

装载一个菜单

有许多方法可以将一个菜单连接到一个窗口上。可以将一个菜单和 Windows 类中的所有窗口建立联系，或者将不同的菜单连接到创建的每一个窗口上。首先，我们讨论一下如何将一个菜单和 Windows 类本身建立联系。

在 Windows 类的定义中，有一行代码可以定义菜单：

```
Winclass.lpszMenuName = NULL;
```

你所要做的工作就是将它赋值为该菜单资源的名称，好，下面就是如何赋值：

```
Winclass.lpszMenuName = "MainMenu";
```

如果“MainMenu”是一个常量的话，可以按下面方式做：

```
Winclass.lpszMenuName = MAKEINTRESOURCE(MainMenu);
```

这样做惟一的一个问题就是创建的每个窗口都会有这样一个相同的菜单。要解决该问题，可以在创建菜单过程中通过传递菜单句柄来将一个菜单指定给一个窗口。但是，要使用菜单句柄，必须使用 LoadMenu()装载菜单资源。下面是LoadMenu()的原型：

```
HMENU LoadMenu(HINSTANCE hInstance, // handle of application instance
LPCTSTR lpMenuName); // menu name string or menu-resource identifier
```

如果函数调用成功的话，LoadMenu()向菜单资源返回一个 HMENU 句柄，这样就可以使用了。

下面是标准的 CreateWindow()函数调用，将菜单“MainMenu”装载到该菜单句柄参数中：

```
// create the window
if (!{hwnd = CreateWindowEx( NULL,    // extended style
                             WINDOW_CLASS_NAME,  // class
                             " Sound Resource Demo ",  // title
                             WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                             0,0,  // initial x,y
                             400,400  // initial width, height
                             NULL,    // handle of parent
                             LoadMenu(hinstance, "MainMenu"),  // handle to menu
                             hinstance,  // instance of this application
                             NULL)))    // extra creation parms

return( 0 );
```

如果 **MainMenu** 是一个常数，调用就和下面相似：

```
LoadMenu(hinstance, MAKEINTRESOURCE(MainMenu)),  // handle to menu
```

提 示

你可能会认为我太关注于资源是通过字符串还是通过符号常量来定义的区别了。但考虑到该问题是导致 Windows 程序员自我毁灭的最多的原因，我认为它值得做更多的工作，不是吗？

当然，在.RC 文件中可以有許多不同的菜单，因此可以将一个不同的菜单连接到每一个窗口上。

将一个菜单连接到每一个窗口的最后一个方法是调用函数 **SetMenu()**：

```
BOOL SetMenu(HWND hwnd, // handle of window to attach to
             HMENU hMenu); // handle of menu
```

SetMenu()采用窗口句柄和菜单句柄（从 **LoadMenu()**中获取），直接将该菜单连接到一个窗口上。这个新菜单将优先于任何以前连接的菜单。下面是一个例子的清单，假设 Windows 类定义该菜单为 **NULL**，如同调用 **CreateWindow()**的菜单句柄一样：

```
// first fill in the window class structure
winclass.cbSize = Sizeof(WNDCLASSEX);
winclass.style  = CS_DBLCLKS | CS_OWNDC |
                  CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
winclass.cbClsExtra   = 0;
winclass.cbWndExtra   = 0;
winclass.hInstance    = hinstance;
winclass.hIcon        = LoadIcon(hinstance,
                                   MAKEINTRESOURCE(ICON_T3DX));
winclass.hCursor      = LoadCursor(hinstance,
                                   MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL; // note this is null
```

```

winclass.lpszClassName = WINDOW_CLASS_NAME;
winclass.hIconSm = LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));
// register the window class
if (!RegisterClassEx(&winclass))
    return(0);

// create the window
if (!(hwnd = CreateWindowEx(NULL,           // extended style
    WINDOW_CLASS_NAME,    // class
    "Menu Resource Demo", // title
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0,0                   // initial x,y
    400,400               // initial width,height
    NULL,                 // handle to parent
    NULL,                 // handle to menu, note it's null
    hinstance,            // instance of this application
    NULL)))               // extra creation parms
    return(0);

// Since the window has been created you can
// attach a new menu at any time

// load the menu resource
HMENU hmenuhandle = LoadMenu(hinstance, "MainMenu");

// attach the menu to the window
SetMenu(hwnd, hmenuhandle);

```

使用第二种方法创建菜单和将该菜单连接到一个窗口上的例子（也就是说在该窗口的创建调用过程中），可以参见 CD-ROM 上的 DEMO3_3.CPP 和相关的可执行文件 DEMO3_3.EXE，图 3.10 表示了 DEMO3_3.EXE 正在运行的情况。



图 3.10 正在运行 DEMO3_3.EXE

这里惟一需要注意的两个文件是资源文件和头文件：DEMO3_3RES.H 和 DEMO3_3.RC。

DEMO3_3RES.H 的内容：

```
// defines for the top level menu FILE
#define MEUN_FILE_ID_OPEN    1000
#define MEUN_FILE_ID_CLOSE   1001
#define MEUN_FILE_ID_SAVE    1002
#define MEUN_FILE_ID_EXIT    1003

// define for the top level menu HELP
#define MEUN_HELP_ABOUT      2000
```

DEMO3_3.RC 的内容：

```
#include "DEMO3_3RES.H"

MainMenu MENU DISCARDABLE
{
    POPUP "File"
    {
        MENUITEM "Open", MENU_FILE_ID_OPEN
        MENUITEM "Close", MENU_FILE_ID_CLOSE
        MENUITEM "Save", MENU_FILE_ID_SAVE
        MENUITEM "Exit", MENU_FILE_ID_EXIT
    } // end POPUP

    POPUP "Help"
    {
        MENUITEM "About", MENU_HELP_ABOUT
    } // end popup

} // end top level menu
```

要编译自己的 DEMO3_3.CPP 可执行文件，一定要确认包含下面文件：

DEMO3_3.CPP——主要源程序。

DEMO3_3RES.H——符号头文件源程序。

DEMO3_3.RC——资源脚本程序。

尝试运行 DEMO3_3.EXE 和相关源程序。改变菜单项，通过向.RC 文件中添加更多的 POPUP 块来添加更多的菜单。并且尝试创建一个层叠式的菜单树。（提示：对于创建菜单的 MENUITEM 来说，只是用一个 POPUP 块来替换 MENUITEM。）

对菜单事件消息的响应

DEMO3_3.EXE 的惟一问题是它不能做任何事情。的确，其中主要问题就是不知道如何侦测菜单项选择和操作产生的消息。本节将主要讨论这个主题。

当滑过一级菜单项时，Windows 菜单系统产生大量的消息（如图 3.11 所示）。

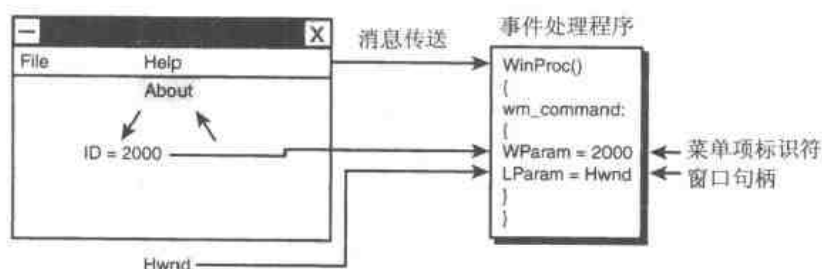


图 3.11 Windows 菜单选择消息流

我们感兴趣的是在选中一个菜单项后放开鼠标时的消息传递。这指的是一个选择过程。选择过程将一个 WM_COMMAND 消息传递到该菜单连接的窗口 WinProc() 函数中。指定的菜单项标识符和其他各种数据存储在消息的 wParam 和 lParam 中，如下所示：

msg——WM_COMMAND。

lparam——传递消息的窗口句柄。

wparam——选中的菜单项标识符。

提示

从技术角度上讲，应当从 wParam 中提取低位的 WORD，以保证 LOWORD() 宏安全。该宏是标准包括中的一部分，因此必须要阅读该内容。

因此你只需要调用函数 switch() 来处理 wParam 的参数值，菜单中定义的另一种 MENUITEM 标识符，就是你的工作了。例如，使用 DEMO3_3.RC 文件中定义的菜单，还应当添加 WM_COMMAND 消息句柄，WinProc() 以下面方式结束：

```
LRESULT CALLBACK WindowProc(HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    // this is the main message handler of the system
    PAINTSTRUCT ps;    // used in WM_PAINT
    HDC          hdc;    // handle to a device context

    // what is the message
    switch(msg)
```

```

{
case WM_CREATE:
    {
    // do initialization stuff here

    // return success
    return(0);
    } break;

case WM_COMMAND:
    {
    switch(LOWORD(wparam))
        {
        // handle the FILE menu
        case MENU_FILE_ID_OPEN:
            {
            // do work here
            } break;
        case MENU_FILE_ID_CLOSE:
            {
            // do work here
            } break;
        case MENU_FILE_ID_SAVE:
            {
            // do work here
            } break;
        case MENU_FILE_ID_EXIT:
            {
            // do work here
            } break;

        // handle the HELP menu
        case MENU_HELP_ABOUT:
            {
            // do work here
            } break;

        default: break;

        } // end switch wparam

    } break; // end WM_COMMAND

case WM_PAINT:
    {
    // simply validate the window
    hdc = BeginPaint(hwnd,&ps);
    // you would do all your painting here
    EndPaint(hwnd,&ps);
    }

```

```

        // return success
        return(0);
    } break;

    case WM_DESTROY:
    {
        // kill the application, this sends a WM_QUIT message
        PostQuitMessage(0);

        // return success
        return(0);
    } break;

    default:break;

} // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wParam, lParam));

} // end WinProc

```

就是如此简单，但这其实是非法的！当然还有其他的操作一级菜单和菜单项本身的消息，可以阅读 Win32 SDK 帮助来获得更多的信息。（我几乎无需了解一个菜单项是否被单击以外的信息。）

作为菜单工作内容的现实的例子，我已经创建了一个很不错的声音演示程序，可以允许通过主菜单来结束一个程序，演示四种不同端口声音效果中的一个，最后通过“Help”菜单弹出一个 Help 对话框。并且.RC 文件中包含声音、图表和光标资源。该程序就是 DEMO3_4.CPP，让我们首先看一下其资源脚本程序和头文件。

DEMO3_4RES.H 的内容：

```

// defines for sounds resources
#define SOUND_ID_ENERGEIZE 1
#define SOUND_ID_BEAM 2
#define SOUND_ID_TELEPORT 3
#define SOUND_ID_WARP 4

// defines for icon and cursor
#define ICON_T3DX 100
#define CURSOR_CROSSHAIR 200

// defines for the top level menu FILE
#define MENU_FILE_ID_EXIT 1000

// defines for play sound top level menu

```

```

#define MENU_PLAY_ID_ENERGIZE      2000
#define MENU_PLAY_ID_BEAM          2001
#define MENU_PLAY_ID_TELEPORT      2002
#define MENU_PLAY_ID_WARP          2003

// defines for the top level menu HELP
#define MENU_HELP_ABOUT            3000

```

DEMO3_4.RC 的内容:

```

#include "DEMO3_4RES.H"

// the icon and cursor resource
ICON_T3DX      ICON    t3dx.ico
CURSOR_CROSSHAIR CURSOR crosshair.cur

// the sound resources
SOUND_ID_ENERGIZE  WAVE energize.wav
SOUND_ID_BEAM      WAVE beam.wav
SOUND_ID_TELEPORT  WAVE teleport.wav
SOUND_ID_WARP      WAVE warp.wav

// the menu resource
SoundMenu MENU DISCARDABLE
{
    POPUP "&File"
    {
        MENUITEM "&Exit", MENU_FILE_ID_EXIT
    } // end popup

    POPUP "&PlaySound"
    {
        MENUITEM "Energize!", MENU_PLAY_ID_ENERGIZE
        MENUITEM "Beam Me Up", MENU_PLAY_ID_BEAM
        MENUITEM "Engage Teleporter", MENU_PLAY_ID_TELEPORT
        MENUITEM "Quantum Warp Teleport", MENU_PLAY_ID_WARP
    } // end popup

    POPUP "Help"
    {
        MENUITEM "About", MENU_HELP_ABOUT
    } // end popup
} // end top level menu

```

在资源脚本程序和头文件（必须包含在主程序中）的基础上，来看一下 DEMO3_4.CPP

装载每一种资源的代码选录。首先，主菜单、图标和光标的装载：

```
winclass.hCursor = LoadCursor(hinstance,
                               MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.lpszMenuName = "SoundMenu";
winclass.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));
winclass.hIconSm= LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));
```

现在就是最有趣的部分——运行每一种声音以及结束菜单项句柄和演示帮助菜单下的关于对话框的 WM_COMMAND 消息的处理。为了简便起见，只将 WM_COMMAND 消息句柄演示一下，因为我们到现在已经阅读了完整的 WinProc()。

```
case WM_COMMAND:
{
    switch(LOWORD(wparam))
    {
        // handle the FILE menu
        case MENU_FILE_ID_EXIT:
        {
            // terminate window
            PostQuitMessage(0);
        } break;

        // handle the HELP menu
        case MENU_HELP_ABOUT:
        {
            // pop up a message box
            MessageBox(hwnd, "Menu Sound Demo",
                      "About Sound Menu",
                      MB_OK | MB_ICONEXCLAMATION);
        } break;
        // handle each of sounds
        case MENU_PLAY_ID_ENERGIZE:
        {
            // play the sound
            PlaySound(MAKEINTRESOURCE(SOUND_ID_ENERGIZE),
                     hinstance_app, SND_RESOURCE | SND_ASYNC);
        } break;
        case MENU_PLAY_ID_BEAM:
        {
            // play the sound
            PlaySound(MAKEINTRESOURCE(SOUND_ID_BEAM),
                     hinstance_app, SND_RESOURCE | SND_ASYNC);
        } break;
        case MENU_PLAY_ID_TELEPORT:
        {
            // play the sound
```

```

        PlaySound(MAKEINTRESOURCE(SOUND_ID_TELEPORT),
                  hinstance_app, SND_RESOURCE | SND_ASYNC);
    } break;
    case MENU_PLAY_ID_WARP:
    {
        // play the sound
        PlaySound(MAKEINTRESOURCE(SOUND_ID_WARP),
                  hinstance_app, SND_RESOURCE | SND_ASYNC);
    } break;

    default: break;
} // end switch wparam
} break; // end WM_COMMAND

```

上面就是所有的内容。

如上所述，资源可以进行大量的工作，并且使用起来很有趣。现在我们暂时中断资源方面的内容，简单介绍一下 WM_PAINT 消息和基本 GDI 操作。

图形设备接口 GDI 介绍

迄今为止，惟一的一次使用 GDI 是在主事件句柄中的 WM_PAINT 消息处理过程中。你应该知道 GDI，也就是图形设备接口，在没有开始使用 DirectX 时是怎样在 Windows 环境下画所有的图形。实际上你还没有学会如何应用 GDI 在屏幕上画任何东西，但这是非常关键的，因为在屏幕上渲染是编写一个视频游戏的最重要的部分。从根本上说一个游戏只是驱动一个视频显示的逻辑过程。本节中将再次讨论 WM_PAINT 消息，包括一些基本的视频概念，以及了解如何在窗口中画一个文本。下一章将重点介绍 GDI。

理解 WM_PAINT 消息对于标准的 GDI 图形和 Windows 编程来讲是非常重要的，因为大多数 Windows 程序的显示都涉及到该消息。在 DirectX 游戏中，就不是这样了，因为 DirectX 或者更专业的 DirectDraw 或 Direct3D 都可以绘制图形，但是仍然需要了解编写 Windows 应用程序的 GDI。

再次出现 WM_PAINT 信息

当窗口的用户区需要刷新时，WM_PAINT 消息就传递到该窗口的 WinProc() 中。到现在为止，还没有对该事件进行过任何处理。下面是你所使用的标准的 WM_PAINT 句柄：

```

PAINTSTRUCT ps; // used in WM_PAINT
HDC  hdc; // handle to a device context

Case WM_PAINT:

```

```

{
// simply validate the window
hdc = BeginPaint(hwnd, &ps);
// you would do all your painting here
EndPaint(hwnd, &ps);
// return success
return(0);
} break;

```

参考图 3.12 来看一下下面的解释。当一个窗口被移动、改变大小或被其他窗口或事件以一定的图形方式变暗时，该窗口的用户区的一部分或全部都需要刷新。发生该事件时，就要传递 WM_PAINT 消息，并且必须处理该消息。

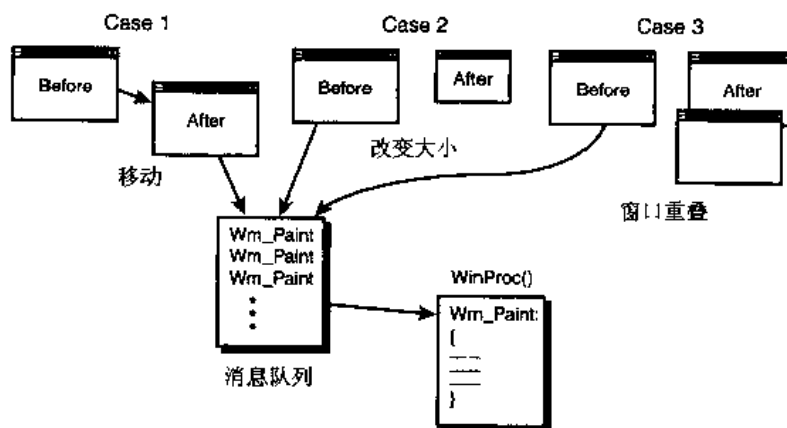


图 3.12 WM_PAINT 消息

就上述程序代码而言，调用 `BeginPaint()` 和 `EndPaint()` 函数可以完成一系列任务。首先，它们使用户区有效；第二，用该窗口创建时 Windows 类定义的背景刷来填充该窗口的背景。

现在你就可以在 `BeginPaint()`—`EndPaint()` 的调用过程中来完成图形操作。但还有一个问题是：只能访问实际上需要刷新的该窗口用户区的一部分。无效矩形区域的坐标都保存在 `BeginPaint()` 函数返回值 `ps` (`PAINTSTRUCT`) 的 `rcPaint` 字段中：

```

typedef struct tagPAINTSTRUCT
{
    HDC hdc;                // graphics device context
    BOOL fErase;            // if TRUE then you must draw background
    RECT rcPaint;           // the RECT containing invalid region
    BOOL fRestore;          // internal
    BOOL fIncUpdate;        // internal
    BYTE rgbReserved[32];   // internal
} PAINTSTRUCT;

```

要刷新内存，下面是 `RECT` 的定义：

```
typedef struct _RECT
{
    LONG left;           // left edge of rectangle
    LONG top;            // upper edge of rectangle
    LONG right;          // right edge of rectangle
    LONG bottom;         // bottom edge of rectangle
} RECT;
```

换句话说，参考图 3.12，该窗口是 400×400，但是只有该窗口的下半部分（300，300—400，400）区域需要重新绘制。因此，通过 `BeginPaint()` 函数调用返回的图形设备内容只是对该窗口的 100×100 的区域有效。很明显，如果要访问整个用户区的话，这就是一个问题了。

该问题的解决方法是直接访问并处理该窗口的图形设备描述表，而不是将其作为一个窗口刷新信息的一部分并通过函数 `BeginPaint()` 传递。使用 `GetDc()` 函数总可以获得一个窗口的图像描述表或 `hdc`，代码如下所示：

```
HDC GetDc(HWND hwnd); // handle of window
```

你只要向访问的图形设备描述表传递窗口句柄，该函数就返回一个指向该窗口的句柄。如果该函数不成功，则返回 `NULL`。在处理完该图形设备描述表句柄后，必须通过调用 `ReleaseDc()` 函数令其返回 Windows，如下所示：

```
int ReleaseDc(HWND hwnd, // handle of window
              HDC hdc);  // handle of device context
```

`ReleaseDc()` 采用该窗口句柄，也就是上述调用 `GetDc()` 函数获得的该设备描述表句柄。

注意



Windows-speak 在处理图形设备描述表时，可能会引起混乱。从技术角度讲，设备描述表句柄可以指向多个输出设备。例如，一个设备描述表可能是一个打印机。因此我通常将只有图像的设备描述表称为图形设备描述表。但数据类型则是 `HDC`，或者指向设备描述表的句柄。因此一般而言，我将一个图形设备描述表变量定义为 `HDC hdc`，而有时我也使用 `HDC gdc`，因为对我来讲它更有意义。在任何一种情况下，都要注意在本书中，图形设备描述表和设备描述表是可以互换的，并且名字为 `hdc` 和 `gdc` 的变量是同一类型的变量。

下面是如何使用 `GetDc()`—`ReleaseDc()` 来处理图形：

```
HDC gdc = NULL; // this will hold the graphics device context

// get the graphics context for the window
if (!(gdc = GetDc(hwnd)))
    error();
// use the gdc here and do graphics—you don't know how yet!

// release the dc back to Windows
ReleaseDc(hwnd, gdc);
```


当然现在还不知道如何处理图形，我将在其他地方讨论它。现在重要的问题就是要了解处理 WM_PAINT 消息的其他方法。另外还有一个问题，就是当调用 GetDc()—ReleaseDc() 时，Windows 并不知道已经使该窗口的用户区恢复或有效。换句话说，如果使用 GetDc()—ReleaseDc() 代替 BeginPaint()—EndPaint()，还会出现新的问题有待解决。

该问题就是 BeginPaint()—EndPaint() 向 Windows 传递一个消息，指令该窗口内容已经恢复（甚至在没有任何图形调用的情况下）。Windows 不会传递任何其他 WM_PAINT 消息。另一方面，如果在 WM_PAINT 句柄中应用 GetDc()—ReleaseDc() 替代了 BeginPaint()—EndPaint()，WM_PAINT 消息将一直不停地传递下去，为什么呢？因为必须使该窗口有效。

要想使需要重画的窗口区域有效，并且通知 Windows 已经恢复了该窗口，应当在调用 GetDc()—ReleaseDc() 之后，再调用 BeginPaint()—EndPaint()，但这样做效率太低。可以使用专用函数，即 ValidateRect()。

```
BOOL ValidateRect(HWND hWnd, // handle of window
CONST RECT *lpRect);        // address of validation rectangle coordinates
```

要使一个窗口有效，将该窗口的句柄连同 lpRect 中的有效区域一同传递。大多数情况下，有效区域应当是整个窗口。因此在 WM_PAINT 句柄中使用 GetDc()—ReleaseDc()，如下所示：

```
PAINTSTRUCT ps; // used in WM_PAINT
HDC  hdc; // handle to a device context
RECT rect; // rectangle of window

Case WM_PAINT:
{
// simply validate the window
hdc = GetDc(hWnd);
// you would do all your painting here
ReleaseDc(hWnd, hdc);

// get client rectangle of window—use Win32 call
GetClientRect(hWnd, &rect);
// validate window
ValidateRect(hWnd, &rect);
// return success
return(0)
} break;
```

提示



注意 GetClientRect() 函数的调用。所做的一切只是获取用户矩形区域坐标。请记住，由于窗口可以任意移动，一个窗口都有两套坐标：Windows 坐标和用户坐标。Windows 坐标相对于屏幕，而用户坐标相对于该窗口的左上角 (0, 0)。图 3.13 更清楚地表示了这种情况。

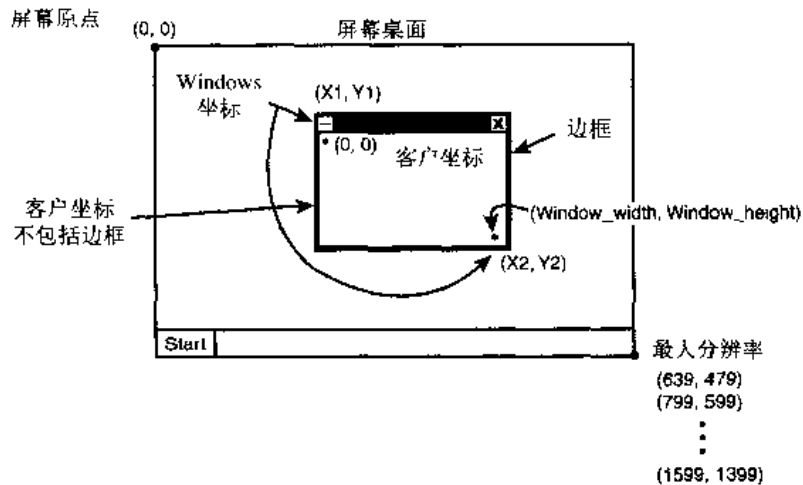


图 3.13 Windows 坐标和用户坐标对比

你可能会说：“有这么麻烦吗？”事实的确如此，因为这是 Windows 系统。请记住，在 WM_PAINT 消息句柄这样做的根本原因就是确认能够在该窗口的用户区域任何位置绘制图形。如果在一个完全无效的窗口上应用 GetDc()—ReleaseDc() 或 BeginPaint()—EndPaint()，那就只有一种可能。但是我们一直在追求两全其美境界，并且我们已差不多做到了这一点。我想告诉你的最后一个技巧就是如何手动使一个窗口无效。

如果想在 WM_PAINT 句柄中以某种方式使整个窗口无效，应当确保由 BeginPaint() 返回的 ps PAINTSTRUCT 的 rcPaint 字段以及相关的 hdc 应当允许访问该窗口的整个用户区域。要做到这一点的话，就可以通过调用 InvalidateRect() 手动放大任何窗口的无效区域，如下所示：

```
BOOL InvalidateRect(HWND hwnd, // handle of window with
                        // changed update region
CONST RECT *lpRect, // address of rectangle coordinates
BOOL bErase); // erase-background flag
```

如果 bErase 为真，调用函数 BeginPaint() 以背景刷填充；否则，就不操作。

在调用 BeginPaint()—EndPaint() 之前只调用 InvalidateRect()，然后在调用 BeginPaint() 时，该无效区域将反映无效区内容和需要使用 InvalidateRect() 添加的内容。大多数情况下，将 InvalidateRect() 的 lpRect 参数设定为 NULL，这样将使整个窗口无效，下面就是其代码：

```
PAINTSTRUCT ps; // used in WM_PAINT
HDC hdc; // handle to a device context

Case WM_PAINT:
{
    // invalidate the entire window
```

```

    InvalidateRect(hwnd, NULL, FALSE);
    // begin painting
    hdc = BeginPaint(hwnd, &ps);
    // you would do all your painting here
    EndPaint(hwnd, &ps);
    // return success
    return(0)
} break;

```

本书大多数情况下，将使用 `GetDc()`—`ReleaseDc()` 来替换其他的 `WM_PAINT` 消息，`BeginPaint()`—`EndPaint()` 只用于 `WM_PAINT` 句柄中。现在讨论一些简单的图形，至少能够输出文本。

视频显示基础和色彩

关于视频现实基础和色彩，我想花一点时间来讨论一下 PC 机上的和图形与色彩有关的一些概念和术语。首先看下面的定义：

- **像素**——光栅显示器（如计算机显示器）上的单个的可寻址的图元。
- **分辨率**——显卡支持的像素数，如 640×480 、 800×600 等等。分辨率越高，图像质量越好，但也需要越多的内存。表 3.2 列出了一些最常用的分辨率和各自的内存需求。
- **色深**——代表屏幕上每一个像素的位数或字节数——每像素的位数 (bpp)。例如，如果每一个像素由 8 位（一个字节）来表示，将只能支持显示 256 色彩，因为 $2^8=256$ 。而如果每一个像素由 16 位（两个字节）来表示，则每个像素可以支持 16384 或者 2^{16} 色彩。并且，色深越高，细节越清晰，占用内存也越多。8 位模式是常用的调色板颜色（将简要介绍），16 位模式称为增强色，24 位和 32 位模式称为真彩色。
- **隔行扫描/逐行扫描显示**——扫描电子枪一次所画的计算机显示的一行称为扫描线符。标准电视对每个图形画两次画面。一个画面包括全部的奇数扫描行，另一个画面包括全部的偶数扫描行。这两个画面连续绘制时，你的眼睛被它们混淆到一起，就创建了一个图形。这仅对于移动的图形是可以接受的，而对于像 Windows 显示这样的静态图形就不能适应了。一些显示卡在隔行扫描时只能支持高分辨率模式。当进行隔行扫描时，显示时一般可以看到闪烁或颤动。
- **视频 RAM**——视频卡上的单板内存数量，代表屏幕上或 texture 内存中的视频图像。
- **刷新速率**——每秒钟视频图像刷新的次数，以 Hz 或 fps（每秒的帧数）来计量。60Hz 是当前最小的可接受的水平，一些显示器或显示卡对于一个实体显示可以支持到 100Hz。
- **2D 加速**——视频卡上的硬件支持，能够协助 Windows 或/和 DirectX 进行 2D 操作，如位图图形、线、圆、文本和图像缩放等等。

- **3D 加速**——视频卡上的硬件支持，能够协助 Windows 或 DirectX/Direct3D 进行 3D 图形渲染。

图 3.14 显示了这些元素。

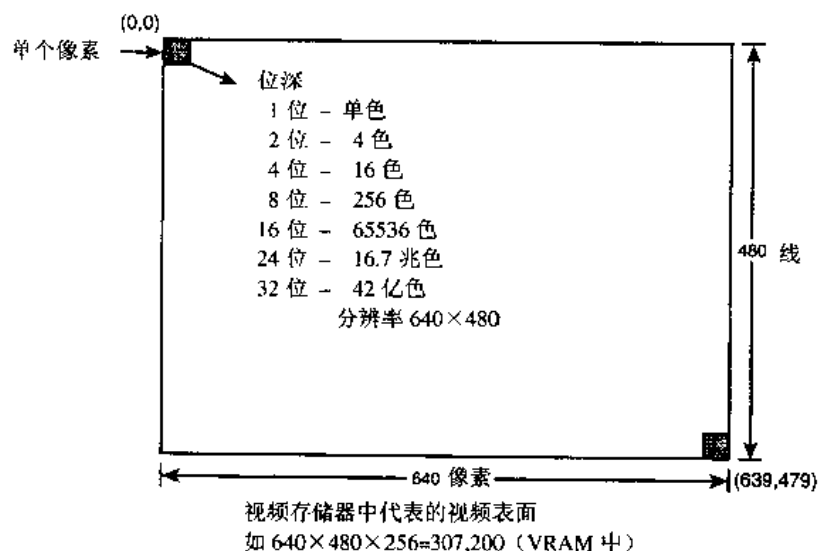


图 3.14 视频显示的机制

表 3.2 视频分辨率和内存需求

分辨率	每像素的位数	内存 (最小—最大)
320×200*	8	64KB
320×240*	8	64KB
640×480	8、16、24、32	307KB~1.22MB
800×600	8、16、24、32	480KB~1.92MB
1024×768	8、16、24、32	786KB~3.14MB
1280×1024	8、16、24、32	1.31MB~5.24MB
1600×1200	8、16、24、32	1.91MB~7.68MB

* 这些称为 X 模式，你的视频卡可能不支持该模式。

当然，表 3.2 只是一些可能的视频模式和色深的例子。你的视频卡可能支持更多的模式。要知道 2MB~4MB 的视频 RAM 轻易就会被消耗掉。好在编写的大多数 DirectX Windows 游戏可以在 320×240 或 640×480 下运行，它们主要依赖于色深，只要一个 2MB 的显卡就可以支持。

RGB 和调色模型

在显卡上表现颜色有两种方式：直接法和间接法。直接色彩模式，或者是 RGB 模式以

代表颜色的红、绿、蓝三原色的 16、24 或 32 位来表示屏幕上的每一个像素（如图 3.15 所示）。这大概是由于红、绿、蓝三原色的添加混色本质造成的。

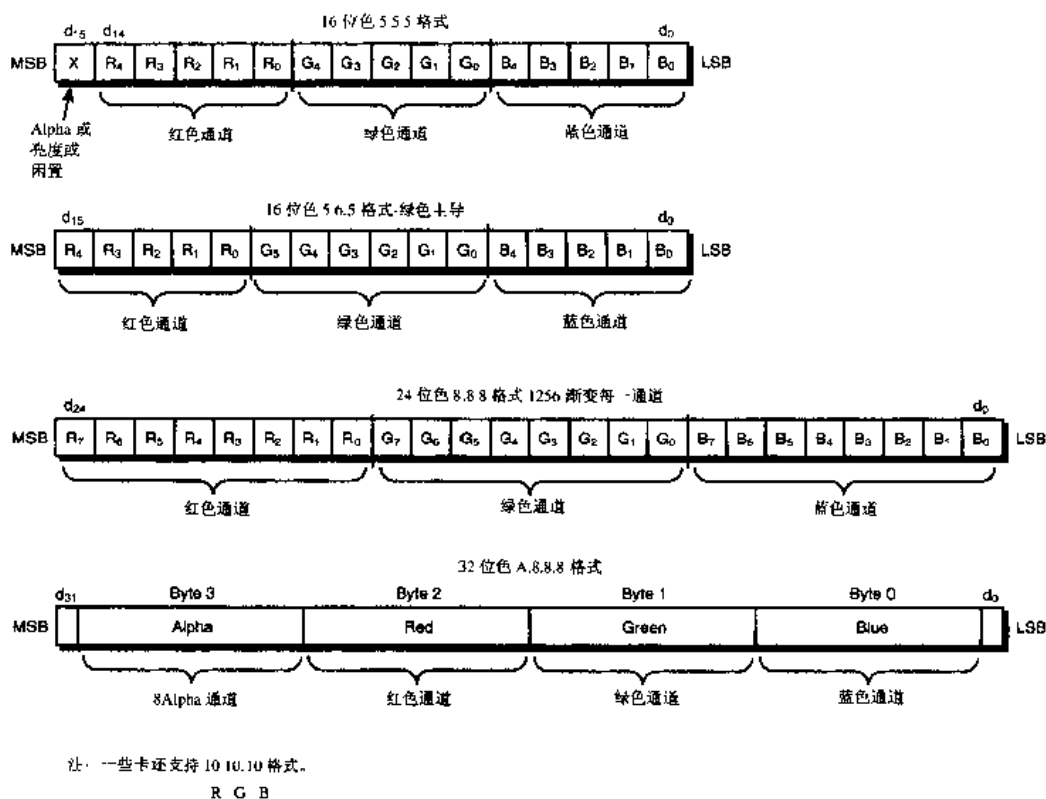


图 3.15 RGB 模式的颜色编码

参考图 3.15，可以看到，对于每一种可能的色深（16、24、32），都由大量的位指定给每一个彩色通道。当然，像 16 位和 32 位彩色，这些数都不能被 3 整除；因此，每个彩色通道中都有一个不等的数。例如，16 位色彩模式就有三种不同的 RGB 编码方式：

- **RGB (6.5.5)** ——6 位红色、5 位绿色和 5 位蓝色。
- **RGB (1.5.5.5)** ——1 位 alpha、5 位红色、5 位绿色和 5 位蓝色。
- **RGB (5.6.5)** ——5 位红色、6 位绿色和 5 位蓝色。以我的经验，这是一种最普通的方式。

24 位色彩模式中各彩色通道均为 8 位。但是 32 位模式很古怪，大多数情况下是 alpha（透明度）、红色、绿色和蓝色通道各 8 位。

基本上说，RGB 模式能够控制屏幕上每一个像素中的红绿蓝的准确成分。调色板模式以间接方式工作。每个像素只有 8 位，可以指定 3 位红色、3 位绿色、2 位蓝色或是三原色的组合。这样就使每种基本颜色只有一点色调变化，并且也很不生动。在这里，8 位模式使用一个调色板。

如图 3.16 所示，一个调色板就是一个 256 输入项的表格，每一个输入项对应一个单字

节数值——从 0 到 255。但是实际上每一个输入项都是由三个 8 位的红绿蓝输入项构成的。从本质上说它是一个全 RGB24 位描述符。颜色查找表工作原理如下：当从屏幕上阅读到一个 8 位颜色模式的像素时，其值为 26，则 26 就用作进入颜色表的一个索引。颜色描述符索引 26 的 24 位 RGB 的值用来驱动传递到显示屏上的实际颜色的红绿蓝通道。以这种方式，在屏幕上可以立即获得 256 色，但是它们可以是来自于 16.7M 色彩或 24 位 RGB 值。图 3.16 表示了该查找过程。

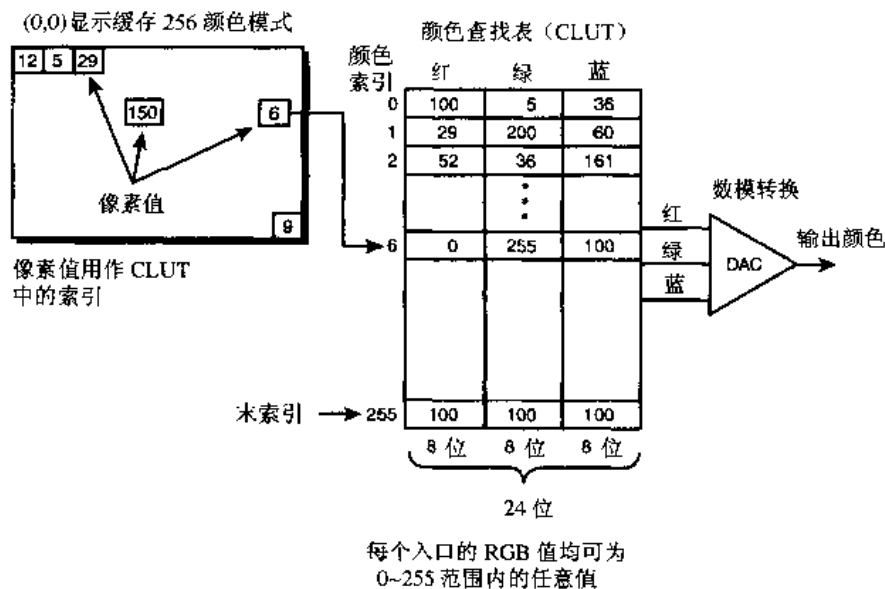


图 3.16 256 色调色板模式工作原理

关于色彩内容我们讨论得有点超前了，但是我希望让你自己先接触一点这些概念，这样在讨论 DirectDraw 的时候，就不会因初次接触它们而显得陌生。实际上标准 GDI 基础的 Windows 图形中，颜色是一个很复杂的问题，以至于 Windows 无论如何都要将颜色提取到 24 位模型。那样在编程时就不必担心色深度的细节等问题。当然，如果你的确对此非常关心，可能会获得更好的结果，但这并不必要。

基本文本打印

Windows 具有我所见到的所有操作系统中最复杂和最强化的文本递交系统。当然，尽管对于大多数游戏程序员来说，所要做的全部工作只是打印分数，但是拥有这个系统还是一件好事。

实际上，GDI 文本引擎对于在一个实时游戏中打印文本太慢了，以至于最后还是要自己来设计 DirectX 基础的文本引擎。现在，先了解一些有关 GDI 打印文本的问题。至少还有助于调试和使用宏输出。

打印文本有两个常用函数：TextOut() 和 DrawText()。TextOut() 是一个很少应用的文本

输出版本，DrawText()应用则非常广泛。我经常使用 TextOut()，是因为它运行很快，并且我不需要 DrawText()的所有细节，但是我们对这两个函数都了解一下。下面是 TextOut()和 DrawText()的原型：

```

BOOL TextOut(HDC hdc, // handle of device context
             int nXStart, // x-coordinate of starting position
             int nYStart, // y-coordinate of starting position
             LPCTSTR lpString, // address of string
             int cbString); // number of characters in string

int DrawText(HDC hdc, // handle to device context
             LPCTSTR lpString, // pointer to string to draw
             int nCount, // string length, in characters
             LPRECT lpRect, // ptr to bounding RECT
             UINT uFormat); // text-drawing flags
    
```

大多数参数都是自解释的。对于 TextOut()来讲，只要传递设备描述表、要打印的 x、y 坐标、ASCII 字符串以及字符串字节长度。而 DrawText()就要稍微麻烦一点。因为它要设定文字包装和格式，采用不同的方法，通过一个着色的矩形区域来打印。因此，DrawText()不使用 x、y 作为开始打印的位置，它采用一个矩形区来定义窗口中开始打印的位置（见图 3.17）。除了使用参数 rect 指出打印区域外，还要传递描述如何打印的一些标志（例如左对齐）。请参考 Win32 关于所有有关标志的文档，其中有大量的标志。我将继续使用 DT_LEFT，该标志最直观，并且调整所有文本左对齐。

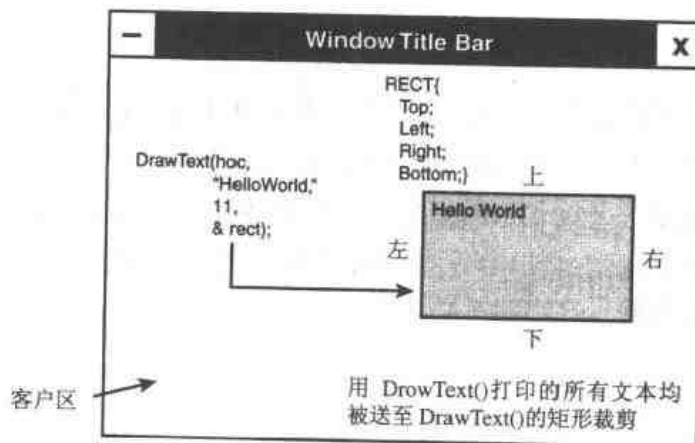


图 3.17 DrawText()正在绘制的矩形区

上面所讨论的函数调用问题中没有涉及颜色。这几乎和 Boogie Nights 一样古怪，但是谁去关心呢？幸运的是，已经有了设定文本前景颜色和背景颜色以及文本透明度模式的方法。

文本的透明度模式指示字符如何绘制。字符是要粘贴到矩形区域还是作为一个覆盖层一个像素一个像素地绘制呢？图 3.18 表示了和打印有关的透明度。如图所示，当文本以透

明模式打印时，看上去就像直接画在一个图像的上面一样。没有透明度的话，只能看到在每个字符周围都是不透明的实体，这样使一切东西都很模糊，非常难看。

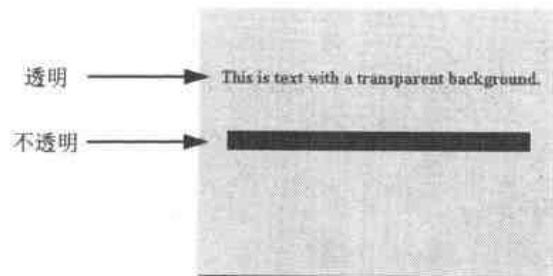


图 3.18 不透明和透明模式的文本打印

提示

无透明模式文本提交速度要更快一些，因此如果在单色背景下打印的话，可以这样做。

让我们看一下设定文本前景和背景颜色的函数：

```
COLORREF SetTextColor(HDC hdc, // handle of device context
    COLORREF Color); // foreground character color
```

```
COLORREF SetBkColor(HDC hdc, // handle of device context
    COLORREF Color); // background color
```

这两个函数都采用图形设备描述表（来自于 `GetDc()` 或 `BeginPaint()` 的调用）以及用于 `COLORREF` 格式中的颜色。一旦设定了这些颜色，这些颜色就一直在变化，直到改变它们为止。另外，当设定了这些颜色后，每个函数都返回当前值，因此在做完后或应用程序结束后，可以保存原来的值。

现在就要准备打印了，但是首先还要了解一下 `COLORREF` 类，下面是 `COLORREF` 类的定义：

```
typedef struct tagCOLORREF
{
    BYTE bRed;        // the red component
    BYTE bGreen;      // the green component
    BYTE bBlue;       // the blue component
    BYTE bDummy;      // unused
} COLORREF;
```

`COLORREF` 在内存中表示为 `0x00bbggrr`。请记住，PC 机就是 Little Endian——也就是说，从低字节到高字节。要创建一个有效的 `COLORREF`，可以使用 `RGB()` 宏，如下所示：


```
COLORREF red      RGB(255,0,0);
COLORREF yellow = RGB(0,255,255);
```

我们看到颜色描述符结构的同时，也就看到 **PALETTEENTRY**，因为它们两个完全相同：

```
typedef struct tagPALETTEENTRY
{
    BYTE peRed;      // red bits
    BYTE peGreen;    // green bits
    BYTE peBlue;     // blue bits
    BYTE peFlags;    // control flags
} PALETTEENTRY;
```

peFlags 可以采用表 3.3 中的值。大多数情况下，应当使用 **PC_NOCOLLAPSE** 和 **PC_RESERVED**，但是现在只要了解它们就可以了。我想要指出的有意思的事情是 **COLORREF** 和 **PALETTEENTRY** 的相似性。除了最后一个字节的解释不同之外，其他完全相同。因此在多数情况下它们是可以相互替换的。

表 3.3 PALETTEENTRY 标志

值	描 述
PC_EXPLICIT	规定逻辑调色板入口的低阶字指定一个硬件调色板索引（高级）
PC_NOCOLLAPSE	规定颜色放置在系统调色板的新入口，而不是和系统调色板已有颜色相匹配
PC_RESERVED	规定调色板动画使用逻辑调色板入口。该标志避免其他窗口和调色板入口颜色一致，因为该颜色是一直在变化的。如果一个新的系统调色板入口可用，该颜色放置在该入口。否则该颜色不能用于动画

下面讨论一下透明度和如何设置透明度。用来设定透明度的函数是 **SetBkMode()**，下面是其原型：

```
int SetBkMode(HDC hdc,          // handle to device context
              int iBkMode);     // transparency mode
```

该函数获取图形设备描述表和新的透明度模式，可能是 **TRANSPARENT** 或者是 **OPAQUE**。该函数返回一个原来的模式，以便于以后的恢复。

好了，现在完全结束了，开始讨论如何打印文本：

```
COLORREF old_fcolor, // old foreground text color
```

```

        old_bcolor; // old background text color

int old_tmode; // old text transparency mode

// first get a graphics device context
HDC hdc = GetDc(hwnd);

// set the foreground color to green and save old one
old_fcolor = SetTextColor(hdc, RGB(0,255,0));

// set the background color to black and save old one
old_bcolor = SetBkColor(hdc, RGB(0,0,0));

// finally set the transparency mode to transparent
old_tmode = SetBkMode(hdc, TRANSPARENT));

// draw some ttext at(20, 30)
TextOut(hdc, 20, 30, "Hello", strlen("Hello"));

// now restore everything
SetTextColor(hwnd, old_fcolor);
SetBkColor(hwnd, old_bcolor);
SetBkMode(hwnd, old_tmode)

// release the device context
ReleaseDc(hwnd, hdc);

```

当然并没有要求必须恢复为原来的值，这样做只是为了说明怎样恢复原来的值。只要该设备描述表的句柄存在，颜色和透明度设置就有效。如果想在绿色文本之外还要画一些蓝色文本，只要将文本颜色改变为蓝色，然后画该文本，而不必重新全部设置所有的值。

请参阅采用前述的技术来打印文本的例子——DEMO3_5.CPP 和可执行文件 DEMO3_5.EXE。该演示程序创建了任在屏幕任意位置随意移动的文本字符串的显示，如图 3.19 所示。

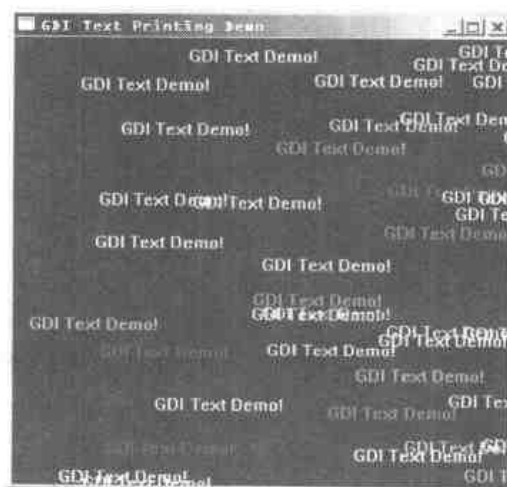


图 3.19 任意位置文本字符串的显示

下面是从该程序的 WinMain () 中摘录的一部分, 可进行所有的动作:

```
// get the dc and hold it
HDC hdc = GetDC(hwnd);

// enter main event loop, but this time we use PeekMessage()
// instead of GetMessage() to retrieve messages
while(TRUE)
{
    // test if there is a message in queue, if so get it
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
        // test if this is a quit
        if (msg.message == WM_QUIT)
            break;

        // translate any accelerator keys
        TranslateMessage(&msg);

        // send the message to the window proc
        DispatchMessage(&msg);
    } // end if

    // main game processing goes here

    // set the foreground color to random
    SetTextColor(hdc, RGB(rand()%256,rand()%256,rand()%256));

    // set the background color to black
    SetBkColor(hdc, RGB(0,0,0));

    // finally set the transparency mode to transparent
    SetBkMode(hdc, TRANSPARENT);

    // draw some text at a random location
    TextOut(hdc,rand()%400,rand()%400,
    "GDI Text Demo!", strlen("GDI Text Demo!"));

    } // end while

// release the dc
ReleaseDC(hwnd,hdc);
```

作为打印文本的第二个例子, 让我们编制一个响应 WM_PAINT 消息的即时更新的计数器, 下面是其程序代码:

```
char buffer[80] // used to print string
static int wm_paint_count = 0; // track number of msg's
```

```

case WM_PAINT:
{
    // simply validate the window
    hdc = BeginPaint(hwnd,&ps);

    // set the foreground color to blue
    SetTextColor(hdc, RGB(0,0,255));
    // set the background color to black
    SetBkColor(hdc, RGB(0,0,0));
    // finally set the transparency mode to transparent
    SetBkMode(hdc, OPAQUE);

    // draw some text at (0,0) reflecting number of times
    // wm_paint has been called
    sprintf(buffer,"WM_PAINT called %d times    ", ++wm_paint_count);
    TextOut(hdc, 0,0, buffer, strlen(buffer));

    EndPaint(hwnd,&ps);
    // return success
    return(0);
} break;

```

请参阅 CD-ROM 上的程序 DEMO3_6.CPP 及其可执行文件 DEMO3_6.EXE。注意，除非移动或重写该窗口，否则将不会打印任何文本。这是因为只有在存在一定原因——恢复或重画该窗口（如移动或改变大小）的时候，才会产生 WM_PAINT 消息。

基本打印工作大概就是这些了。当然，DrawText()函数可以做更多的工作，但这已经超出了要求。还可以浏览一下字体以及其他全部和打印有关的内容，但这些通常都用于完整的 WindowsGUI 编程，本书中不予讨论。

处理重要事件

如前面所学，Windows 是一个事件驱动的操作系统。对事件发生反应是标准 Windows 程序的最重要的一个方面。下面部分主要讨论使用窗口操作、输入设备和时钟来处理的一些重要事件。如果你已经能够处理这些基本事件，就可以从 Windows 知识库中学习更多的知识，来处理任何可能在编写 DirectX 游戏时碰上的问题。DirectX 游戏本身和时间以及 Windows 操作系统关系不大。

Window 操作

Windows 传递大量的消息来通知用户操作其窗口。表 3.4 列出了一些 Windows 生成的

很有意思的操作消息。

表 3.4 Windows 操作消息

值	说 明
WM_ACTIVATE	当一个窗口被激活或者取消时传递。该消息首先传递到被撤销的最高级窗口的窗口程序中
WM_ACTIVATEAPP	当属于一个应用程序的窗口而不是其他活动窗口可能要被激活时传递。该消息向被激活和被取消的窗口程序都传递
WM_CLOSE	当一个窗口或一个应用程序终止时作为一个信号传递
WM_MOVE	当窗口移动后传递
WM_MOVING	向用户移动的窗口传递。通过处理该消息，应用程序能够检测到拖动的矩形区域的大小和位置，并且在需要时还可以改变其大小和位置
WM_SIZE	当一个窗口改变大小后传递
WM_SIZING	向用户改变大小的窗口传递。通过处理该消息，应用程序能够检测到改变大小的矩形区域的大小和位置，并且在需要时还可以改变其大小和位置

先来看一下 WM_ACTIVATE、WM_CLOSE、WM_SIZE 和 WM_MOVE 及其工作原理。对于每个消息，我都将列出该消息、wparam、lparam 和一些说明以及 WinProc() 处理程序对于该事件的简短的处理实例。

消息：WM_ACTIVATE

参数：

```
fActive      = LOWORD(wParam);           // activation flag
fMinimized   = (BOOL)HIWORD(wParam);     // minimized flag
hwndPrevious = (HWND)lParam;             // window handle
```

fActive 参数主要定义了该窗口发生了什么事，即该窗口是被激活还是被取消。该信息保存在 wparam 的低阶字中，可以取表 3.5 中所示的值。

表 3.5 WM_ACTIVATE 的激活标志

值	说 明
WA_CLICKACTIVE	通过鼠标单击激活该窗口
WA_ACTIVE	通过除鼠标之外的工具（如键盘）激活该窗口
WA_INACTIVE	取消该窗口

fMinimized 变量只能指示该窗口是否最小化。如果该变量非零则为真。hwndPrevious 值根据 fActive 参数的值来标识该窗口是被激活还是被取消。如果 fActive 的值为

WA_INACTIVE, hwndPrevious 是被激活窗口句柄；如果 fActive 的值为 WA_ACTIVE 或者 WA_CLICKACTIVE, hwndPrevious 则是被取消窗口句柄。该句柄可以为 NULL。

实际上如果想要知道应用程序什么时间被激活，可以使用 WM_ACTIVATE 消息。如果用户每次都不使用 Alt+Tab 组合键处理应用程序，或者是使用鼠标来选择其他的应用程序，该消息是很有用的。而当应用程序重新被激活时，你可能想播放一段音乐或其他操作。但无论是什么操作，这都已经超出了目前内容的范围。

下面是在 WinProc()主处理程序中的如何激活应用程序的代码：

```
case WM_ACTIVATE:
{
    // test if window is being activated
    if (LOWORD(wparam)!=WA_INACTIVE)
    {
        // application is being activated
    } end if
else
{
    // application is being deactivated
} end else

}break;
```

消息：WM_CLOSE

参数：无

WM_CLOSE 消息非常酷。该消息在 WM_DESTROY 和 WM_QUIT 传递之前被发送。WM_CLOSE 消息指示用户关闭窗口。如果 WinProc()值返回 0，则什么也不会发生，并且用户也不能关闭窗口！请参阅 DEMO3_7.CPP 及其可执行文件 DEMO3_7.EXE 来查看操作过程。请尝试关闭该程序——可能做不到！

警告



当不能关闭 DEMO3_7.EXE 时不必惊慌。只要同时按下 Ctrl+Alt+Del，将出现任务管理器。然后选择终止 DEMO3_7.EXE 应用程序。就可以停止了——就像硅谷中以 F 开头的电子商店的服务一样。

下面是 DEMO3_7.CPP 中位于 WinProc()中的空 WM_CLOSE 句柄代码：

```
case WM_CLOSE:
{
    // kill message, so no further WM_DESTROY is sent
    return(0);
} break;
```

如果使用户疯狂是你的目标的话，上述代码便可以达到目的。一个更好的方法是使用一个消息框来捕获 WM_CLOSE 消息以证实应用程序是否将关闭亦或正在进行工作。

DEMO3_8.CPP 及其可执行文件就是依照这个思路工作的。当尝试关闭该窗口时，会出现一个消息框询问是否确定要关闭该窗口。该过程的逻辑流如图 3.20 所示。

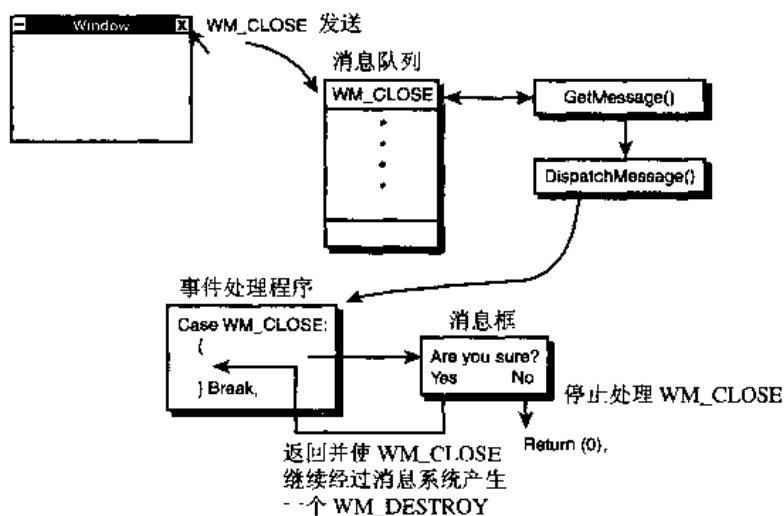


图 3.20 WM_CLOSE 的逻辑流

下面的 DEMO3_8.CPP 中处理 WM_CLOSE 消息的部分代码：

```

case WM_CLOSE:
{
// display message box
int result = MessageBox(hwnd,
    "Are you sure you want to close this application?",
    "WM_CLOSE Message Processor",
    MB_YESNO | MB_ICONQUESTION);

// does the user want to close?
if (result == IDYES)
{
// call default handler
return (DefWindowProc(hwnd, msg, wParam, lParam));
} // end if
else // throw message away
return(0);

} break;

```

注意默认消息句柄 DefWindowProc() 的调用。当用户回答 YES 并且想继续标准关闭过程的时候调用 DefWindowProc()。如果已经知道做法，可以传递一个 WM_DESTROY 消息，但是由于现在还没有学习如何传递消息，只能调用默认句柄。总之哪一种方法都可以。

下面我们看一下 WM_SIZE 消息，如果已经编写了一个窗口游戏，并且用户要求经常改变视窗的话，该消息就非常重要了！

消息: WM_SIZE

参数:

```
fwSizeType = wParam;           // resizing flag
nWidth      = LOWORD(lParam);  // width of client area
nHeight     = HIWORD(lParam);  // height of client area
```

fwSizeType 标志指示将会发生哪种尺寸的改变, 如表 3.6 所示, lParam 客户区宽度和高度指示新窗口用户区的尺寸。

表 3.6 WM_SIZE 的改变尺寸标志

值	说 明
SIZE_MAXHIDE	当有一些窗口最大化时, 该消息传递给所有的弹出的窗口
SIZE_MAXIMIZED	该窗口最大化
SIZE_MAXSHOW	当有一些窗口恢复到上一次尺寸时, 该消息传递给所有的弹出的窗口
SIZE_MINIMIZED	该窗口最小化
SIZE_RESTORED	既不使用 SIZE_MINIMIZED 的值, 也不使用 SIZE_MAXIMIZED 的值来改变窗口的大小

如我所说, 处理 WM_SIZE 消息对于窗口游戏非常重要, 因为当窗口尺寸改变时, 必须调整图像显示来适应。但是如果游戏在全屏状态下运行时就不必调整图像显示, 但是在窗口游戏中, 用户可能会改变窗口的大小。当用户改变窗口大小时, 应当使显示对中, 并且调整环境, 或者使用任何方法保证图像准确。作为一个跟踪 WM_SIZE 消息的实例, DEMO3_9.CPP 在窗口尺寸改变时打印出该窗口的尺寸。DEMO3_9.CPP 跟踪 WM_SIZE 消息的代码如下所示:

```
case WM_SIZE:
{
    // extract size info
    int width = LOWORD(lParam);
    int height = HIWORD(lParam);

    // get a graphics context
    hdc = GetDC(hwnd);

    // set the foreground color to green
    SetTextColor(hdc, RGB(0,255,0));

    // set the background color to black
    SetBkColor(hdc, RGB(0,0,0));

    // set the transparency mode to OPAQUE
    SetBkMode(hdc, OPAQUE);
```



```
// draw the size of the window
sprintf(buffer,
"WM_SIZE Called - New Size = (%d,%d)", width, height);
TextOut(hdc, 0,0, buffer, strlen(buffer));

// release the dc back
ReleaseDC(hwnd, hdc);

} break;
```

警告



应当知道 WM_SIZE 消息句柄的程序存在一个潜在的问题：当一个窗口的尺寸改变时，不仅传递 WM_SIZE 消息，同时 WM_PAINT 消息也同时传递。因此，如果传递 WM_SIZE 消息后传递 WM_PAINT 消息，WM_PAINT 中的程序代码将擦除背景，因此在 WM_SIZE 消息中打印该信息。幸运的是，并不存在这种情况，但是这是一个当消息发生故障或者不按次序传递的问题出现时的很好的实例。

下面，让我们看一下 WM_MOVE 消息。该消息和 WM_SIZE 消息几乎完全相同，但是该消息是在窗口移动时而不是尺寸改变时传递。下面是其详细内容：

消息：WM_MOVE

参数：

```
xPos = (int) LOWORD(lParam); // new horizontal position in screen coords
yPos = (int) HIWORD(lParam); // new vertical position in screen coords
```

WM_MOVE 消息在一个窗口移动到一个新位置时传递，如图 3.21 所示。该消息在该窗口移动之后传递，而不是在实际移动过程中传递。如果想跟踪一个窗口的每个像素的精确移动，你就需要处理 WM_MOVING 消息。在多数情况下，直到用户已完成窗口的移动才停止该处理过程。

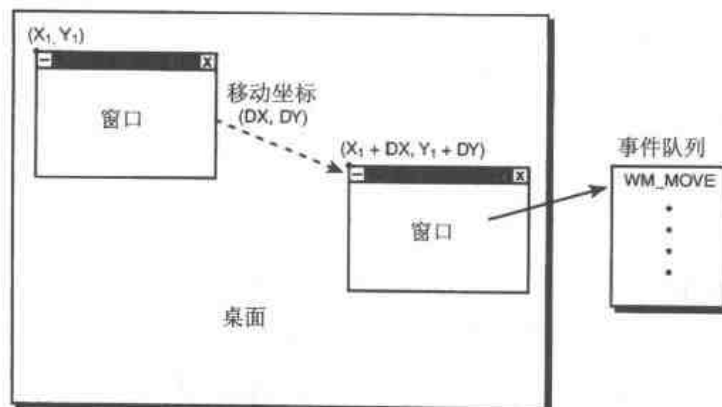


图 3.21 WM_MOVE 消息的产生

DEMO3_10.CPP 及其可执行文件 DEMO3_10.EXE 给出了跟踪一个窗口移动的实例，当窗口移动时，打印该窗口的新的位置。下面是处理 WM_MOVE 过程的程序代码：

```
case WM_MOVE:
{
    // extract the position
    int xpos = LOWORD(lparam);
    int ypos = HIWORD(lparam);

    // get a graphics context
    hdc = GetDC(hwnd);

    // set the foreground color to green
    SetTextColor(hdc, RGB(0,255,0));

    // set the background color to black
    SetBkColor(hdc, RGB(0,0,0));

    // set the transparency mode to OPAQUE
    SetBkMode(hdc, OPAQUE);

    // draw the size of the window
    sprintf(buffer,
    "WM_MOVE Called - New Position - (%d,%d)", xpos, ypos);
    TextOut(hdc, 0,0, buffer, strlen(buffer));

    // release the dc back
    ReleaseDC(hwnd, hdc);

    } break;
```

以上就是窗口操作消息的内容。显然，还有好多相关的内容，但是现在已经了解了其诀窍。要记住的就是 Windows 中有可以做任何事情的消息。如果想了解更多内容，请参阅 Win32 帮助。而且可以肯定的是：你完全可以找到所需要的消息。

下一部分将讨论输入设备，这样就可以和用户（或你自己）进行交流，并且建立更多的有趣的演示和试验，来帮助你精通 Windows 编程。

击打键盘

很久很久以前，使用键盘简直是不可想像的事。要使用键盘，必须首先编写一个中断处理程序，创建一个状态表，并且使用大量的有趣的技巧。我只是一个水平很低的程序员，但是我可以毫不遗憾地说，我一点也没有忘记编写键盘处理程序。

现在可以使用 `DirectInput` 来访问键盘、鼠标、游戏操作杆以及其他的输入设备。毋庸置疑，还要学会如何使用 `Win32` 库来访问键盘和鼠标。即使没有其他用途，你仍然需要键盘和鼠标等设备来响应 GUI 交互作用和/或在本书中创建更迷人的演示，直到我们学习了 `DirectInput` 为止。因此不要再啰嗦了，让我们看一下键盘的工作原理。

键盘是由许多键、微控制器和电子设备支持构成。当按下键盘上的一个或多个键时，一系列的数据流将传递给 Windows，描述按下的该键（或组合键）。Windows 随后处理这些数据流，向你的窗口传递键盘事件消息。令人高兴的是在 Windows 环境下，可以以多种方式访问该键盘消息。

- 通过 `WM_CHAR` 消息
- 通过 `WM_KEYDOWN` 和 `WM_KEYUP` 消息
- 通过调用 `GetAsyncKeyState()`

每种方法的工作方式都有所不同。`WM_CHAR` 和 `WM_KEYDOWN` 消息在按下键盘上的键或发生事件时由 Windows 产生。但是在两个消息中所封装的信息类型有所不同。当你按下一个键盘上的键（如 A）时，将产生两串数据：

- 扫描码
- ASCII 码

扫描码是一个指定给键盘上每一个键的单值码，和 ASCII 码无关。许多情况下，你只想了解是否按下了 A 键，而并不关心 Shift 键是否也同时按下等等。基本上说，只是将键盘作为一种临时转换器来使用。这通过扫描码就可以完成。当按下键时，产生扫描码由 `WM_KEYDOWN` 消息来负责。

而 ASCII 码是熟码（cooked）数据。这表示如果按下键盘上的 A 键，而不按下 Shift 键或不使用 Caps Lock 键，将只能看到一个字符 a。如果按下 Shift+A，可以看到一个字符 A。`WM_CHAR` 可以传递这种消息。

可以使用上述任何一种技术——该内容已经超过了范围。例如，如果想编写一个文字处理器的话，你可能会使用 `WM_CHAR` 消息，因为该字符事关重大，所以要求必须是 ASCII 代码，而非虚拟的扫描码。另一方面，如果你正在编写一个游戏，并且 F 表示开枪、S 表示猛刺、Shift 键表示保护，那谁去关心 ASCII 码是什么？你可能只想知道键盘上的指定按钮是否已被按下。

阅读键盘的最后一种方法就是使用 `Win32` 函数 `GetAsyncKeyState()`，该函数跟踪状态表中的该键的最后已知状态——和 `BOOL` 型（逻辑）转换的阵列一样。这是我更喜欢使用的方法，因为你不必编写一个键盘处理器。

现在对每一种方法都了解了一点，让我们按顺序依次对每一种方法进行详细讨论，首先从 `WM_CHAR` 消息开始：

`WM_CHAR` 消息具有下面参数：

`wparam`——包含所按下键的 ASCII 码。

`lparam`——包含一个位编码状态矢量，描述其他的特定的可能被按下的控制键。该位解

码过程如表 3.7 所示。

表 3.7 键状态矢量的位编码

位	说 明
0~15	包含重复计数，用户按键的重复次数
16~23	包含扫描码。该值依赖于初始设备制造厂家
24	布尔型；扩展键标识符。如果为 1，该键是一个扩展键，例如加强型的 101 或 102 键盘上的右边的 Alt 和 Ctrl 键
29	布尔型；指示是否按下 Alt 键
30	布尔型；指示前一个键的状态。这是个无用的键
31	布尔型；指示键的转换状态。如果结果为 1，该键被释放；否则该键一直按住

要处理 WM_CHAR 消息，你要做的就是为它编写一个消息句柄，如下所示：

```
case WM_CHAR:
{
// extract ascii code and state vector
int ascii_code = wParam;
int key_state = lParam;

// take whatever action

} break;
```

当然，你可以测试一些非常有意思的不同状态的信息。例如，下面是如何测试按下 Alt 键的例子：

```
// test the 29th bit of key_state to see if it's true

#define ALT_STATE_BIT 0x20000000
if (key_state & ALT_STATE_BIT)
{
// do something
} //end if
```

也可以使用相似的逐位测试和操作来测试其他状态。

作为处理 WM_XCHAR 消息的例子，我已经创建了一个在按住键时以 16 进制格式输出字符和状态矢量的演示程序。该程序是 DEMO3_11.CPP 及可执行文件 DEMO3_11.EXE。试着按住这些组合键，看其输出结果。下面是从 WinProc()中摘录的处理和演示 WM_CHAR 信息的代码：

```

case WM_CHAR:
{
    // get the character
    char ascii_code = wparam;
    unsigned int key_state = lparam;

    // get a graphics context
    hdc = GetDC(hwnd);

    // set the foreground color to green
    SetTextColor(hdc, RGB(0,255,0));

    // set the background color to black
    SetBkColor(hdc, RGB(0,0,0));

    // set the transparency mode to OPAQUE
    SetBkMode(hdc, OPAQUE);

    // print the ascii code and key state
    sprintf(buffer,"WM_CHAR: Character = %c  ",ascii_code);
    TextOut(hdc, 0,0, buffer, strlen(buffer));

    sprintf(buffer,"Key State = 0X%X  ",key_state);
    TextOut(hdc, 0,16, buffer, strlen(buffer));

    // release the dc back
    ReleaseDC(hwnd, hdc);

} break;

```

下一个键盘事件消息 **WM_KEYDOWN** 和 **WM_CHAR** 相似，只是其信息不是“熟码”。在 **WM_KEYDOWN** 消息中传递的键数据是虚拟的该键的扫描码，而不是 ASCII 码。虚拟扫描码和标准扫描码相似，都是由键盘产生的，不同的是虚拟扫描码保证和键盘完全一致。例如，在 101 AT 型键盘上的某一键的扫描码为 67，而在其他厂商生产的键盘上则可能是 69，这种现象也是可能的，看出问题来了吗？

Windows 中使用的方法是将真实的扫描码虚拟化为虚拟扫描码放置在一个察看表中。作为程序员，我们使用虚拟扫描码，用 Windows 进行翻译。下面是 **WM_KEYDOWN** 消息的具体内容：

消息：**WM_KEYDOWN**

wparam——包含所按下键的虚拟键代码。表 3.8 列出了一些最常用的键。

lparam——包含一个位解码状态矢量，描述其他的可能被按下的专用控制键。该位解码过程如表 3.8 所示。

表 3.8 虚拟键代码

符 号	位	说 明
VK_BACK	08	退格键
VK_TAB	09	Tab 键
VK_RETURN	0D	回车键
VK_SHIFT	10	Shift 键
VK_CONTROL	11	Ctrl 键
VK_PAUSE	13	Pause 键
VK_ESCAPE	1B	ESC 键
VK_SPACE	20	空格键
VK_PRIOR	21	页上移键
VK_NEXT	22	页下移键
VK_END	23	END 键
VK_HOME	24	HOME 键
VK_LEFT	25	左箭头键
VK_UP	26	上箭头键
VK_RIGHT	27	右箭头键
VK_INSERT	2D	插入键
VK_DELETE	2E	删除键
VK_HELP	2F	帮助键
No VK_Code	30~39	0~9 键
No VK_Code	41~5A	A~Z 键
VK_F1 - VK_F12	70~7B	F1~F12 键

注意：键 A~Z 和 0~9 没有 VK 代码。必须使用数字常数或是定义自己的虚拟键。

除了 WM_KEYDOWN 消息以外，还有一个 WM_KEYUP 消息。该消息和 WM_KEYDOWN 消息有着一样的参数，即：wparam 含有虚拟键代码，而 lparam 含有键状态矢量。惟一的区别是 WM_KEYUP 在松开一个键时传递。

例如，如果正在使用 WM_KEYDOWN 消息来控制，请看下面程序：

```
case WM_KEYDOWN:
{
// get virtual key code and data bits
int virtual_code = (int)wparam;
int key_state = (int)lparam;
```

```

// switch on the virtual_key code to be clean
switch(virtual_code)
{
    case VK_RIGHT:{ } break;
    case VK_LEFT: { } break;
    case VK_UP:   { } break;
    case VK_DOWN: { } break;
    // more cases...

    default: break;
} // end switch

// tell windows that you processed the message
return(0);
} break;

```

来试验一下修改 DEMO3_11.CPP 中的程序，支持 WM_KEYDOWN 消息而不是支持 WM_KEYUP 消息。做完后我们看一下阅读键盘的最后一种方法。

阅读键盘的最后一种方法是调用一个键盘状态函数：GetKeyboardState()、GetKeyState() 或 GetAsyncKeyState()。我们重点讨论一下 GetAsyncKeyState()，因为该函数只服务于单个键而不是整个键盘，这正是你所关心的。如果对其他函数感兴趣的话，可以在 Win32 SDK 中查阅。GetAsyncKeyState()的原型是：

```
SHORT GetAsyncKeyState(int virtual_key);
```

只能向该函数传递想测试的虚拟键代码，并且如果返回值高位是 1 的话，该键被按下。否则该键被松开。我编写一些宏来使得该函数更容易些。

```

#define KEYDOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEYUP(vk_code)   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

```

使用 GetAsyncKeyState()的好处在于它和事件循环无关。可以在任何时候测试按键。例如，正在编写一个游戏，并且打算跟踪箭头键、空格键或者是 Ctrl 键。根本不必去处理 WM_CHAR 或 WM_KEYDOWN 消息，而只要编写一段类似下面的代码：

```

if (KEYDOWN(VK_DOWN))
{
    // move ship down, whatever
} // end if

if (KEYDOWN(VK_SPACE))
{
    // fire weapons maybe?
} // end if

```

```
// and so on
```

同样道理，想侦测一个键是否松开，可加入下面代码：

```
if (KEYUP(VK_ENTER))
{
    // disengage engines
} //end if
```

我创建了一个连续输出 WinMain()中箭头键状态的演示程序的例子。该程序是 DEMO3_12.CPP，可执行文件为 DEMO3_12.EXE。下面是该程序中的 WinMain()主程序：

```
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpcmdline,
                  int ncmdshow)
{
    WNDCLASSEX winclass; // this will hold the class we create
    HWND        hwnd;    // generic window handle
    MSG          msg;     // generic message
    HDC          hdc;     // graphics device context

    // first fill in the window class structure
    winclass.cbSize      = sizeof(WNDCLASSEX);
    winclass.style       = CS_DBLCLKS | CS_OWNDC |
                          CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra  = 0;
    winclass.cbWndExtra  = 0;
    winclass.hInstance  = hinstance;
    winclass.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor     = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;
    winclass.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);

    // save hinstance in global
    hinstance_app = hinstance;

    // register the window class
    if (!RegisterClassEx(&winclass))
        return(0);

    // create the window
    if (!(hwnd = CreateWindowEx(NULL,                // extended style
                              WINDOW_CLASS_NAME,    // class
                              "GetAsyncKeyState() Demo", // title
                              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
```



```

        0,0,        // initial x,y
        400,300,   // initial width, height
        NULL,      // handle to parent
        NULL,      // handle to menu
        hinstance, // instance of this application
        NULL)))    // extra creation parms
return(0);

// save main window handle
main_window_handle = hwnd;

// enter main event loop, but this time we use PeekMessage()
// instead of GetMessage() to retrieve messages
while(TRUE)
{
    // test if there is a message in queue, if so get it
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
        // test if this is a quit
        if (msg.message == WM_QUIT)
            break;

        // translate any accelerator keys
        TranslateMessage(&msg);

        // send the message to the window proc
        DispatchMessage(&msg);
    } // end if

    // main game processing goes here

    // get a graphics context
    hdc = GetDC(hwnd);

    // set the foreground color to green
    SetTextColor(hdc, RGB(0,255,0));

    // set the background color to black
    SetBkColor(hdc, RGB(0,0,0));

    // set the transparency mode to OPAQUE
    SetBkMode(hdc, OPAQUE);

    // print out the state of each arrow key
    sprintf(buffer,"Up Arrow: = %d  ",KEYDOWN(VK_UP));
    TextOut(hdc, 0,0, buffer, strlen(buffer));

    sprintf(buffer,"Down Arrow: = %d  ",KEYDOWN(VK_DOWN));
    TextOut(hdc, 0,16, buffer, strlen(buffer));
}

```

```

sprintf(buffer, "Right Arrow: = %d  ", KEYDOWN(VK_RIGHT));
TextOut(hdc, 0, 32, buffer, strlen(buffer));

sprintf(buffer, "Left Arrow: = %d  ", KEYDOWN(VK_LEFT));
TextOut(hdc, 0, 48, buffer, strlen(buffer));

// release the dc back
ReleaseDC(hwnd, hdc);

} // end while

// return to Windows like this
return(msg.wParam);

} // end WinMain

```

如果浏览了 CD-ROM 上的整个源程序，将会发现该窗口消息句柄中根本没有 WM_CHAR 和 WM_KEYDOWN 的句柄。在 WinProc() 中处理的消息越少越好。另外，这是首次看到在 WinMain() 中发生的动作，WinMain() 是处理整个游戏过程的一部分。注意到没有任何时间延迟或同步的问题，因此说信息的刷新是自由运行的（也就是说运行足够快）。在第四章“Windows GDI、控制和突发奇想”中，将会学习定时、如何将过程锁定在一定帧速上等等问题。现在让我们讨论一下鼠标。

按住鼠标

鼠标可能是所有的计算机输入设备中最有创造性的一个设备了。点击鼠标，鼠标垫物理反映出屏幕表面——这真是一个创举。如你所想，Windows 具有大量的鼠标的消息，我们仅讨论其中的两类消息：WM_MOUSEMOVE 和 WM_*BUTTON。

首先讨论 WM_MOUSEMOVE 消息。关于鼠标要记住的第一件事情就是，鼠标位置是和其在窗口的用户区的位置相对应的。参考图 3.22，鼠标传递的是相对于窗口左上角（坐标为 0, 0）的坐标。

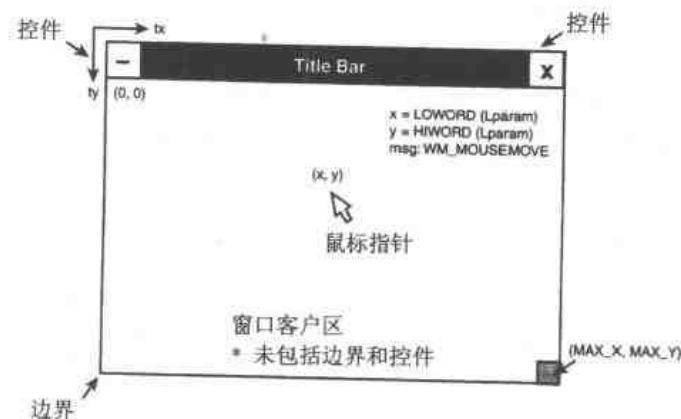


图 3.22 鼠标移动详解

和其他不同的是，WM_MOUSEMOVE 消息是相当直观的。

消息：WM_MOUSEMOVE

参数：

```
int mouse_x = (int)LOWORD(lParam);
int mouse_y = (int)HIWORD(lParam);
int buttons = (int) wParam
```

lparam 中的状态基本上都是以 16 位入口编码的，wparam 中的按键编码如表 3.9 所示。

表 3.9 WM_MOUSEMOVE 的按键位解码

值	说 明
MK_LBUTTON	如果按下鼠标左键则设定该值
MK_MBUTTON	如果按下鼠标中间按键则设定该值
MK_RBUTTON	如果按下鼠标右键则设定该值
MK_CBUTTON	如果按下 Ctrl 键则设定该值
MK_SHIFT	如果按下 Shift 键则设定该值

现在所要做的就是将其中一个位码和按键状态进行逻辑“和”运算，可以检测到哪一个按键被按下。下面是一个跟踪鼠标的位置 x、y 以及左键和右键的例子。

```
case WM_MOUSEMOVE:
{
// get the position of the mouse
int mouse_x = (int)LOWORD(lParam);
int mouse_y = (int)HIWORD(lParam);

// get the button state
int buttons = (int)wParam;

// test if left button is down
if (buttons & MK_LBUTTON)
{
// do something
} // end if

// test if right button is down
if (buttons & MK_RBUTTON)
{
// do something
} // end if

} break;
```

琐碎，太琐碎了。CD-ROM 上的 DEMO3_13.CPP 及其可执行文件 DEMO3_13.EXE 给出了鼠标跟踪的例子。该程序使用上面程序作为开头，输出鼠标的位置和按键状态。请注意在鼠标移动时按键如何变化。这正是所希望的，因为该消息是在鼠标移动时而不是在按键按下时传递的。

下面还有一些细节问题，WM_MOUSEMOVE 并不能保证在任何时间都传递。如果移动鼠标过快则来不及跟踪。因此，不要以为能够很好地跟踪每一次鼠标移动，当然大多数情况下还是没有问题的，但这个问题还是要记住。可能你正在绞尽脑汁地想如果按下一个鼠标按键而没有任何移动如何跟踪鼠标。当然 Windows 具有一整套的鼠标消息，请看表 3.10。

表 3.10 鼠标按键消息

消 息	说 明
WM_LBUTTONDOWNBLCLK	鼠标左键双击
WM_LBUTTONDOWN	按下鼠标左键
WM_LBUTTONUP	松开鼠标左键
WM_MBUTTONDOWNBLCLK	鼠标中键双击
WM_MBUTTONDOWN	按下鼠标中键
WM_MBUTTONUP	松开鼠标中键
WM_RBUTTONDOWNBLCLK	鼠标右键双击
WM_RBUTTONDOWN	按下鼠标右键
WM_RBUTTONUP	松开鼠标右键

在 lparam 和 wparam 中，按键消息也含有鼠标位置信息——该信息被编码保存于 WM_MOUSEMOVE 消息中。例如，要测试左键双击，可以这样做：

```
case WM_LBUTTONDOWNBLCLK:
{
    // extract x,y and buttons
    int mouse_x = (int)LOWORD(lParam);
    int mouse_y = (int)HIWORD(lParam);

    // do something intelligent

    // tell windows you handled it
    return(0);
} // break;
```

太强大了，你认为呢？Windows 简直为我们想到家了。

将消息传递给自己

我想讨论的最后一个题目是自我传递消息。有两种方式实现这项工作：

SendMessage()——直接向窗口传递一个要处理的消息。如果接收窗口已经处理了该消息，在 **WinProc()** 之后该函数返回。

PostMessage()——向窗口传递一个消息序列并且直接返回。如果不在意处理消息的时间延迟，或者该消息优先级较低，可以使用该函数。

这两个函数的原型基本相似，如下所示：

```
LRESULT SendMessage(HWND hWnd, // handle of destination window
                    UINT Msg,   // message to send
                    WPARAM wParam, // first message parameter
                    LPARAM lParam); // second message parameter
```

SendMessage() 的返回值是由传递该消息到的窗口的 **WinProc()** 返回的值。

```
BOOL PostMessage(HWND hWnd, // handle of destination window
                UINT Msg,   // message to post
                WPARAM wParam, // first message parameter
                LPARAM lParam); // second message parameter
```

如果 **PostMessage()** 成功的话，将返回一个非零值。注意，这一点和 **SendMessage()** 不同。因为，**SendMessage()** 实际上调用 **WinProc()**，而 **PostMessage()** 只是将一个消息不经处理就放到接收窗口的消息序列中。

你可能对为什么要向自己传递消息而感到疑惑不解，从字面上说有大量理由。可以说是 Windows 的设计人员希望你这样做，这也是在窗口环境下工作的原理。例如，在下一章中，当我们讨论如按键之类的窗口控制，传递消息是和一个控制窗口对话的惟一途径。如果是我的话，我喜欢内容更具体一点。

在前面所有的演示程序中，已经通过双击关闭对话框或按住 **Alt+F4** 来终止程序。如果能通过程序来关闭演示程序，其实没有什么不好。

WM_CLOSE 或者是 **WM_DESTROY** 能够完成这项工作。如果使用 **WM_CLOSE**，可能只会给应用程序一点提示；而 **WM_DESTROY** 则相对更严谨一些。但是无论哪种方式，都需要像下面这样做：

```
SendMessage(hWnd, WM_DESTROY, 0, 0);
```

或者延迟一下，不在意消息是否编入序列中，可以使用 `PostMessage()`：

```
PostMessage(hwnd, WM_DESTROY, 0, 0);
```

这两种情况都会终止应用程序，除非在 `WM_DESTROY` 句柄中有一个控制逻辑。下一个问题是何时传递消息。这个内容已经超出了范围。在一个游戏中，可以跟踪 `Esc` 键，来终止程序。下面是如何使用主事件循环中的 `KEYDOWN()`宏来终止程序：

```
if (KEYDOWN(VK_ESCAPE))
    SendMessage(hwnd, WM_CLOSE, 0, 0);
```

关于上面的运行程序代码，可以参见 CD-ROM 上的 `DEMO3_14.CPP` 及其可执行文件 `DEMO3_14.EXE`。该程序准确执行了上面程序的逻辑过程。也可以尝试一下使用 `PostMessage()`改变传递到 `WM_DESTROY` 的消息。

警告



向主事件循环外传递消息可能会引起不可预知的问题。例如，在主事件循环外通过 `SendMessage()`向 `WinProc()`直接传递一个消息，可以终止窗口。但是如果按正常规程那样以为主事件循环中的事件句柄会处理该消息，则会产生一个超出执行次序的故障。这表示本以为事件 B 在事件 A 之后发生，但是在某些情况下事件 B 在事件 A 之前发生了。这就是传递消息时容易出现的典型问题，因此一定要确认深思熟虑。`PostMessage()`通常更安全一些，因为它不会跳出事件序列。

最后，还有一种传递自己自定义消息的方式 `WM_USER`。使用 `PostMessage()`或 `SendMessage()`以 `WM_USER` 作为消息类型来传递消息。可以将所有想添入的消息都放到 `wparam` 和 `lparam` 的值中。例如，使用 `WM_USER` 消息可以为内存管理系统创建大量的虚拟消息，如下所示：

```
// define for memory manager
#define ALLOC_MEM      0
#define DEALLOC_MEM    1

// send WM_USER message, use the lparam as amount of memory
// and the wparam as the type of operation
SendMessage(hwnd, WM_USER, ALLOC_MEM, 1000);
```

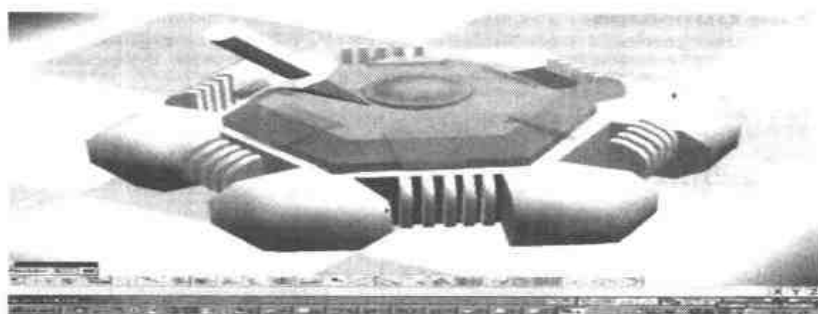
然后，在 `WinProc()`中可以这样做：

```
case WM_USER:
{
// what is the virtual message
switch(wparam)
{
case ALLOC_MEM: { } break;
case DEALLOC_MEM: { } break;
// .. more messages
} // end switch
} break;
```

如上所示，可以对 `wparam` 和 `lparam` 中的任何消息进行编码，或者按上面例子中所示的那样去做，或者去做一些更有意义的事。

总 结

感谢上帝！我还没有忘记应当结束这一章。在本章中我们讨论了资源、菜单、输入设备、GDI 和消息传递。一篇质量很高的关于 Windows 的专题论文大约要 3000 页，这样你就能够了解我进退维谷的处境了吧！但我认为我们已经覆盖了大量有用的内容。本章之后，你就完全可以利用 Windows 来工作了。



4

WindowsGDI、控件和突发奇想

本章是纯粹的 Windows 编程内容的最后一章。我们将详细讨论使用图形设备接口界面的内容。内容包括绘制像素、线和简单的几何形状。然后我们简略讨论一下定时方法，以 Windows 的子控件来结束 Windows 编程内容。最后简要介绍一下 T3D 游戏控制台模板程序，在本书后面所有内容中将该模板程序作为所有演示的开始。下面列出了本章涉及的主要内容：

- ➔ 高级 GDI 编程、笔、画刷和渲染
- ➔ 子控件
- ➔ 系统定时函数
- ➔ 传递消息
- ➔ 获取消息
- ➔ T3D 游戏控制台

高级 GDI 图形

我曾经提到过，GDI 和 DirectX 相比太慢了。但是 GDI 在各方面都很优秀，并且是 Windows 系统内置的渲染引擎，也就是说，如果想创建任何工具或标准的 GDI 应用程序，了解关于 GDI 的工作方式是很有益处的。而了解如何将 GDI 和 DirectX 混合在一起使用则是使用 GDI 功能来模拟 DirectX 程序中未能实现的函数一种好方法。因此 GDI 可以在编写游戏程序演示功能时作为一种速度较慢的软件仿真，至少你还可以了解 GDI 的内容。

下面我们来讨论一下一些基本的 GDI 操作，还可以通过浏览 Win32 SDK 来了解更多的 GDI 内容，在本书中可以学到一些基本的技巧，而不是 GDI 的每一个功能。这就像看一

个计算机分销商展览——看到了一个，也就看到了全部。

图形设备描述表

在第三章“高级 Windows 编程”中，可以经常看到设备描述表的类型句柄，或者是 HDC，即表示设备描述表的句柄的数据类型。这里设备描述表是一个图形设备描述表类型，当然还有其他的设备描述表，如打印机设备描述表。无论怎样讲，可能都对什么是图形设备描述表，以及图形设备描述表到底代表什么意思而感到疑惑不解。这都是很好的问题。

一个图形设备描述表实际上就是对一种安装在系统中的视频图像卡的描述。因此，当访问一个图形设备描述表或句柄时，实际上就表示安装在计算机系统上的视频卡具体描述及其分辨率和色彩容量。对于使用 GDI 的任何图形调用，该信息都是必须的。从本质上说，你所提供的指向任何 GDI 函数的 HDC 句柄，都用来访问一个函数需要操作的视频系统的重要信息。这就是需要一个图形设备描述表的原因所在。

图形设备描述表跟踪编程过程中可能改变的软件设置。例如，GDI 使用大量的图形对象，如笔、画刷、线的类型等等。GDI 使用上述基本数据描述来绘制任何一种图元。因此，尽管当前画笔颜色是你设置的某种颜色，并且也不是视频卡的默认颜色，但是图形设备描述表仍然跟踪它。因此，图形设备描述表不仅是视频系统的硬件描述，而且还是记录和保存设置的信息库，由此你的 GDI 函数调用能够使用这些设置，而不是和这些调用一起发送。这样可以保存 GDI 调用的大量参数。下面让我们看一下如何使用 GDI 对图形渲染。

色彩、画笔和画刷

如果认真考虑的话，能够在计算机屏幕上绘制的对象的类型并没有多少。当然绘制图形的形状和颜色有无穷多种，但对象的类型是很有限的。这些对象就是点、线和矩形。其他的任何东西都是这些基本图元对象类型的组合。

GDI 所采用的这种方法有点像一个画家。一个画家使用颜色、画笔和画刷来绘画——我们也是这样。GDI 以相同的方式工作，并且还有下面的定义：

- **画笔**——用于画线条和轮廓，具有颜色、粗细和线型。
- **画刷**——用于填充任何闭合的对象，具有颜色、样式，甚至可以是位图。图 4.1 给出了一个详细的标签。

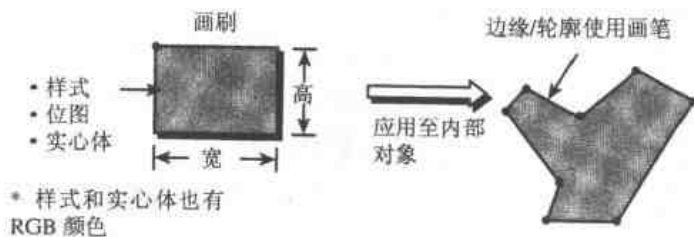


图 4.1 详细标注的画刷

在我们具体接触画笔和画刷以及实际使用它们之前，必须先了解 GDI 的情况。GDI 一般一次只使用一个画笔和一个画刷。在你的系统配置中可以有许多画笔和画刷，但是在当前图形设备描述表中每次只有一个画笔或画刷被激活。这样就必须为图形设备描述表选择对象，以便于使用。

请记住，图形设备描述表不仅是一个视频卡及其工作的描述，而且还是当前绘制工具的描述。画笔和画刷是该设备描述表跟踪的工具中的主要的样本，并且必须从该图形设备描述表中或之外选择这些工具。该过程称之为选定。当程序运行时，将选定一个新的画笔，然后选择画笔输出，也可能选定不同的画刷并且选择画刷输出等等。应当记住的是一旦该内容中选定了—个绘图对象，就要一直使用该对象，直到改变对象为止。

最后，无论何时创建了一个新的画笔或画刷，在完成绘制图形之后必须删除该画笔或画刷。这是非常重要的，因为 Windows GDI 关于画笔和画刷句柄就有如此之多的存取窗口，通常学会 GDI 几乎要耗尽精力，但我们一会儿就可以学会。好，现在我们首先讨论画笔，然后是画刷。

使用画笔

画笔句柄称之为 HPEN。下面是如何创建一个 NULL 画笔：

```
HPEN pen_1 = NULL;
```

pen_1 指示一个画笔句柄，但是，pen_1 仍然不能使用所希望的信息来替代或删除。该操作可以通过下面两种方法中的一种来完成。

- 使用存储对象
- 创建一个用户定义的画笔

请记住，存储对象或者是存储任何东西仅仅是 Windows 所拥有的一些默认样式的对象。对于画笔，已经有许多已经定义的画笔样式，但是这还是有限的样式。可以使用下面所是的 GetStockObject()函数来检索大量的不同的对象句柄，包括画笔句柄、画刷等等。

```
HGDIOBJ GetStockObject(int fnObject); // type of stock object
```

该函数只采用希望的存储对象的样式，返回一个该对象的句柄。画笔的样式是预先定义的存储对象，如表 4.1 所示。

表 4.1 存储对象样式

值	描 述
BLACK_PEN	黑色画笔
NULL_PEN	中空的画笔
WHITE_PEN	白色画笔
BLACK_BRUSH	黑色画刷

续表

值	描 述
DKGRAY_BRUSH	深灰色画刷
GRAY_BRUSH	灰色画刷
HOLLOW_BRUSH	中空的画刷（相当于 NULL_BRUSH）
LIGRAY_BRUSH	淡灰色画刷
NULL_BRUSH	中空的画刷（相当于 HOLLOW_BRUSH）
WHITE_BRUSH	白色画刷
ANSI_FIXED_FONT	标准 Windows 固定间距（等宽）的系统字体
ANSI_VAR_FONT	标准 Windows 可变间距（成比例间距）的系统字体
DEFAULT_GUI_FONT	只用于 Windows 95：用户界面对象如菜单和对话框的默认字体
OEM_FIXED_FONT	由生产商（OEM）确定的固定间距（等宽）字体
SYSTEM_FONT	系统字体，默认情况下，Windows 使用系统字体来绘制菜单、对话框控制功能和文本。在 Windows 3.0 版本之后的系统中，系统字体为成比例间距字体，3.0 以前的 Windows 版本使用等宽系统字体
SYSTEM_FIXED_FONT	Windows 3.0 之前的版本使用的固定间距（等宽）系统字体。该存储对象用来和 Windows 早期的版本兼容

由表 4.1 可以看到，并不能从表 4.1 中选择全部的画笔样式（这只是 GDI 的一点幽默）。下面是如何创建一个白色画笔的例子：

```
HPEN white_pen = NULL;
White_pen = GetStockObject(WHITE_PEN);
```

当然 GDI 并不知道 white_pen，因为并不能将 white_pen 选定到图形设备描述表中，但我们将它选定了。

创建画笔的更有趣的方法是通过定义画笔颜色、线条样式和像素宽度来自己创建画笔。用来创建画笔的函数是 CreatePen()，如下所示：

```
HPEN CreatePen(int fnPenStyle,      // style of the pen
               int nWidth,          // width of pen in pixels
               COLORREF crColor);   // color of pen
```

nWidth 和 crColor 参数非常容易理解，但是 fnPenStyle 需要解释一下。

大多数情况下，都想画实线，但有时可能也需要画一条虚线来表示图标程序中的一些内容。可以通过画大量的被一段空格分开的实线来作为一条虚线，但是为什么不让 GDI 来做这个工作呢？线条样式支持这个功能。当 GDI 表现线条时，进行逻辑“与”运算或者掩

盖住一个线条样式筛选程序。由此，就可以绘制由点和虚线、实像素或者其他任何的一维实体来构成的线条。表 4.2 给出了一些可以从中选用的有效线条样式。

表 4.2 CreatePen() 的线条样式

值	描 述
PS_NULL	画笔不可见
PS_SOLID	画笔为实线
PS_DASH	画笔为虚线
PS_DOT	画笔为点
PS_DASHDOT	画笔为点划线
PS_DASHDOTDOT	画笔为双点划线

例如，我们创建三种画笔，每种画笔都是 1 个像素宽，样式为实线：

```
// the red pen, notice the use of the RGB macro
HPEN red_pen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));

// the green pen, notice the use of the RGB macro
HPEN green_pen = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));

// the blue pen, notice the use of the RGB macro
HPEN blue_pen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
```

下面创建白色的虚线画笔：

```
HPEN white_dashed_pen = CreatePen(PS_DASHED, 1, RGB(255, 255, 255));
```

非常简单吧！下面看一下如何向图形设备描述表中选择画笔。我们仍然不知道如何绘制图形，下面首先看一下这个概念。

要将任何 GDI 对象选择到图形设备描述表，使用 SelectObject() 函数，如下所示：

```
HGDIOBJ SelectObject(HDC hdc, // handle of device context
                     HGDIOBJ hgdiobj); // handle of object
```

SelectObject() 使用图形描述表句柄以及所选择对象的句柄。注意，SelectObject() 是一个多种组合形式的函数，可以使用不同的句柄类型，这是因为所有的图形对象句柄都是数据类型 HGDIOBJ（GDI 对象句柄）的子类。并且，该函数返回当前的从该内容中取消选定的对象句柄。换句话说，如果向该内容中选定了—一个新画笔，很明显就要取消对旧画笔的选定。因此，可以保存旧画笔并且可以一直存储着该画笔。下面是向图形描述表中选定一个画笔并保存旧画笔的一个例子：

```
HDC hdc; // the graphics context, assume valid
```

```
// create the blue
HPEN blue_pen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));

HPEN old_pen = NULL; // used to store old pen

//select the blue pen in and save the old pen
old_pen = SelectObject(hdc, blue_pen);

// do drawing...

// restore the old pen
SelectObject(hdc, old_pen);
```

最后，当已经使用 `GetStockObject()` 或 `CreatePen()` 创建的画笔进行工作之后，必须保存它们。该项工作由 `DeleteObject()` 函数来完成，该函数和 `SelectObject()` 函数相类似，也是一个多种组合形式的函数，可以删除不同的句柄类型。下面是其原型：

```
BOOL DeleteObject(HGDIOBJ hObject); // handle to graphic object
```

警告



当取消画笔时应当非常小心。如果要删除一个当前选定的对象，或者选定一个当前已经删除的对象，将导致发生错误，并且可能是一个 GP 默认值。

下一个问题是对于图形对象何时调用 `DeleteObject()` 函数。代表性地说，可以在程序结束时来进行这个工作。如果创建并使用几百个对象，并且在程序的剩余部分不使用它们，就应当随时删除这些对象。这是因为 Windows GDI 只有有限的资源。下面例子是如何释放和删除在前面例子中创建的一组画笔：

```
DeleteObject(red_pen);
DeleteObject(green_pen);
DeleteObject(blue_pen);
DeleteObject(white_dashed_pen);
```

警告



不要尝试删除已经删除过的对象，这样做将导致无法预料的结果。

使用画刷

下面再讨论一下画刷，除外观外，画刷在很多方面都和画笔相似。画刷常用于填充图

形对象，而画笔则用来绘制对象的轮廓或简单的线。但是所有相同的原则都是在变化的。画刷句柄称之为 **HBRUSH**。要定义一个空白的画刷对象，如下所示：

```
HBRUSH brush_1 = NULL;
```

如果要采用画刷的不同形式，则要通过 **GetStockObject()** 使用表 4.1 中的一种画刷类型，或者自己定义一个画刷类型。例如，下面是如何创建一个浅灰色存储画刷的例子：

```
brush_1 = GetStockObject(LTGRAY_BRUSH);
```

太简单了，不是吗？要创建更有趣的画刷，可以选择填充图案类型和色彩，就像在画笔中所做的工作一样。不幸的是 **GDI** 将画刷规定为两类：纯色的和阴影的。我认为这是非常愚蠢的——**GDI** 应当允许所有的画刷都是带阴影线的，并且只具有纯色类型。创建纯色填充画刷的函数是 **CreateSolidBrush()**，如下所示：

```
HBRUSH CreateSolidBrush(COLORREF crColor); // brush color
```

要创建一个纯绿色的画刷，应当如下所示的做法：

```
HBRUSH green_brush = CreateSolidBrush(RGB(0, 255, 0));
```

要将该画刷选定到图形设备描述表中，可以：

```
HBRUSH old_brush = NULL;
old_brush = SelectObject(hdc, green_brush);
// draw something with brush
// restore old brush
SelectObject(hdc, old_brush);
```

在程序结束时，应当删除该画刷对象：

```
DeleteObject(green_brush);
```

在一种环境下，可以创建、选定、使用并删除一个对象。下面是如何创建带图案的或带阴影线的画刷。

要创建一个带阴影线的画刷，则使用 **CreateHatchBrush()** 函数：

```
HBRUSH CreateHatchBrush(int fnStyle,          // hatch style
                        COLORREF clrref); // color value
```

画刷的样式可以选择表 4.3 中所列的值。

表 4.3 **CreateHatchBrush()** 的样式

值	描 述
HS_BDIAGONAL	向下 5° 的左—右阴影线
HS_CROSS	水平和垂直的交叉阴影线
HS_DIAGCROSS	45° 的交叉阴影线

续表

值	描 述
HS_FDIAGONAL	向上 45° 的左—右阴影线
HS_HORIZONTAL	水平阴影线
HS_VERTICAL	垂直阴影线

最后的画刷例子是创建一个红色交叉阴影线的画刷：

```
HBRUSH red_brush = CreateHatchBrush(HS_CROSS, RGB(255, 0, 0));
```

将它选定到设备描述表中：

```
HBRUSH old_brush = SelectObject(hdc, red_hbrush);
```

最后恢复旧画刷，删除创建的红色画刷：

```
SelectObject(hdc, old_brush);
DeleteObject(red_hbrush);
```

当然我们仍然没有使用画笔或画刷做任何事情，但是我们将会用到它们。

点、线、平面多边形和圆

现在已经理解了画笔和画刷的概念，就应当学习如何使用这些实体在实际程序中来绘制对象。首先看一下最简单的图形对象——点。

直接绘制点

使用 GDI 绘制点是很容易的事，不需要画笔或画刷。这是因为一个点就是一个简单的像素，选定画笔或画刷并不起任何作用。要在窗口用户区中绘制一个点，需要在窗口中使用 HDC 以及要绘制点的坐标和颜色。

但是并不需要选定颜色，只要使用一个包含所有信息的函数调用 SetPixel()，请看：

```
COLORREF SetPixel(HDC hdc, // the graphics context
                  int x,    // x-coordinate
                  int y,    // y-coordinate
                  COLORREF crColor); // color of pixel
```

该函数在窗口中使用 HDC 以及 (x,y) 坐标和颜色，然后绘制上该像素点，并返回实际绘制的颜色。如果以 256 色模式并要求一个并不存在的 RGB 颜色，GDI 将绘制一种和所要求颜色最接近的颜色，然后返回实际绘制的 RGB 颜色。如果对传递到函数中的 (x,y) 坐标的实际意义感到不清楚，请看图 4.2。图中表示了一个窗口以及 Windows GDI 使用的

坐标系，该坐标系是一个颠倒的第一象限的笛卡尔坐标系，也就是说，y 从上向下为正方向。

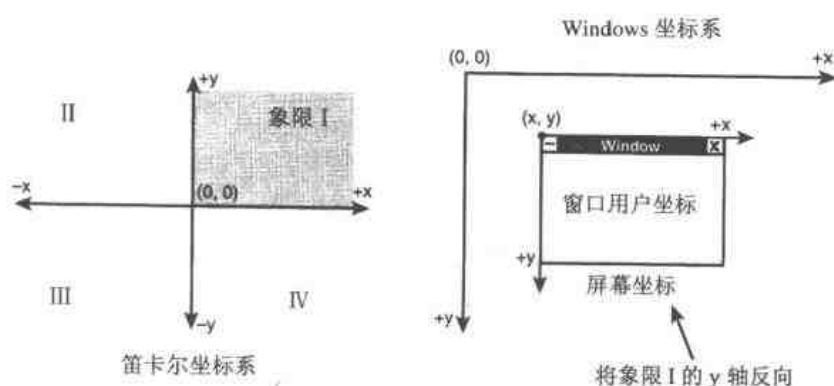


图 4.2 和标准笛卡尔坐标系相对应的 Windows 坐标系

理论上讲，GDI 还有其他的变换模式，但是上述模式为默认模式，并且是用于所有的 GDI 和 DirectX 中的一种模式。请注意，坐标原点 (0,0) 是窗口用户区域的左上角。可以应用 `GetWindowDc()` 而不是 `GetDc()` 来获得整个窗口的 HDC。不同之处在于如果使用 `GetWindowDc()` 来检索 HDC，图形设备描述表则适用于整个窗口。使用 `GetDc()` 来检索 HDC，可以在包括该窗口控制功能在内的区域上面绘制对象，而不仅仅是在用户区。下面的实例是在 400×400 的窗口中绘制 1000 个随机位置和颜色的点：

```
HWND hwnd; //assume this is valid
HDC hdc;   // used to access window

// get the dc for the window
hdc = GetDc(hwnd);

for (int index=0; index<1000; index++)
{
    // get random position
    int x = rand()%400
    int y = rand()%400

    COLORREF color = RGB(rand()%255, rand()%255, rand()%255);
    SetPixel(hdc, x, y, color);
} // end for index
```

绘制像素点的实例请看 DEMO4_1.CPP 以及 DEMO4_1.EXE。这两个程序以连续的循环的形式演示了上面的程序代码。图 4.3 是该程序运行的屏幕结果。

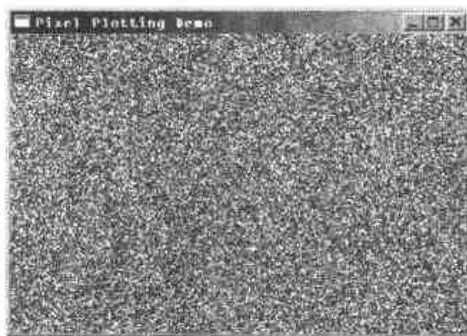


图 4.3 绘制像素点的程序 DEMO4_1.EXE 的演示

绘制线条

下面绘制下一个最基本的对象——线条。要绘制线条，需要创建画笔，然后调用线条绘制函数。在 GDI 环境下，线条要稍微复杂一点。GDI 绘制线条一般采用三个步骤的过程：

1. 创建画笔，并在图形设备描述表中选定画笔。所有的线条都将使用该画笔来绘制。
2. 设定该线条的初始位置。
3. 从起始位置到终点位置绘制线条（该终点位置成为下一段线条的起始位置）。
4. 如果想绘制更多的线条，重复步骤 3。

大致来讲，GDI 具有一个不可见的光标，来跟踪要绘制的线条的当前起始位置。要绘制线条的话，必须自己设定该起始位置，但是一旦设定，GDI 将随着每段线条的绘制而更新，以满足绘制复杂对象（如多边形）的要求。设定该线条光标的初始位置的函数是 `MoveToEx()` 函数：

```
BOOL MoveToEx(HDC hdc, // handle of device context
               int x, // x-coordinate of new current position
               int y, // y-coordinate of new current position
               LPPOINT lpPoint); // address of old current position
```

假定绘制一条从(10,10)到(50,60)的线条，应当首先调用 `MoveToEx()`，如下所示：

```
// set current position
MoveToEx(hdc, 10, 10, NULL);
```

注意最后一个参数为 `NULL`。如果要保存最后一个位置，则：

```
POINT last_pos; // used to store last position

// set current position, but save last
MoveToEx(hdc, 10, 10, &last_pos);
```

顺便再提一下 `POINT` 结构：

```
typedef struct tagPOINT
```

```

    { // pt
      LONG x;
      LONG y;
    } POINT;

```

一旦设定了线条的初始位置，可以调用 `LineTo()` 函数来绘制一段线条：

```

BOOL LineTo(HDC hdc,    // device context handle
            int xEnd,    // destination x-coordinate
            int yEnd);  // destination y-coordinate

```

下面来进行一个完整的绘制线条的实例，从 (10,10) 到 (50,60) 绘制一条绿色实线：

```

HWND hwnd; // assume this is valid

// get the dc first
HDC hdc = GetDc(hwnd);

// create the green pen
HPEN green_pen = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));

// select the pen into the context
HPEN old_pen = SelectObject(hdc, green_pen);

// draw the line
MoveToEx(hdc, 10, 10, NULL);
LineTo(hdc, 50, 60);

// restore old pen
SelectObject(hdc, old_pen);

// delete the green pen
DeleteObject(green_pen);

// release the dc
ReleaseDc(hwnd, hdc);

```

要绘制一个顶点分别为 (20,10)、(30,20)、(10,20) 的三角形，如下面所示：

```

// start the triangle
MoveToEx(hdc, 20, 10, NULL);

// draw first leg
LineTo(hdc, 30, 20);
// draw second leg
LineTo(hdc, 10, 20);

// close it up
LineTo(hdc, 20, 10);

```

这样你就会明白 `MoveToEx()`—`LineTo()` 技术有用的原因了。

绘制线条的实例，请看 `DEMO4_2.CPP`。该程序高速绘制了随机位置的线条。图 4.4 给出了其结果。

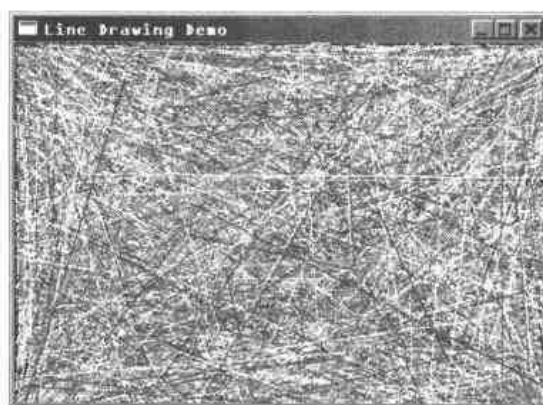


图 4.4 绘制线条程序 DEMO4_2.CPP

绘制矩形

GDI 绘制的下一个内容是矩形。矩形使用画笔和画刷（如果矩形内部区域需要填充的话）来绘制。因此说，矩形是最复杂的 GDI 基本图形单元。要绘制图形，则使用 `Rectangle()` 函数，如下所示：

```

BOOL Rectangle(HDC hdc,      // handle of device context
               int nLeftRect, // x-coord. of bounding
                               // rectangle's upper-left corner
               int nTopRect,  // y-coord. of bounding
                               // rectangle's upper-left corner
               int nRightRect, // x-coord. of bounding
                               // rectangle's lower-right corner
               int nBottomRect, // y-coord. of bounding
                               // rectangle's lower-right corner

```

`Rectangle()` 函数使用当前的画笔和画刷绘制矩形，如图 4.5 所示。

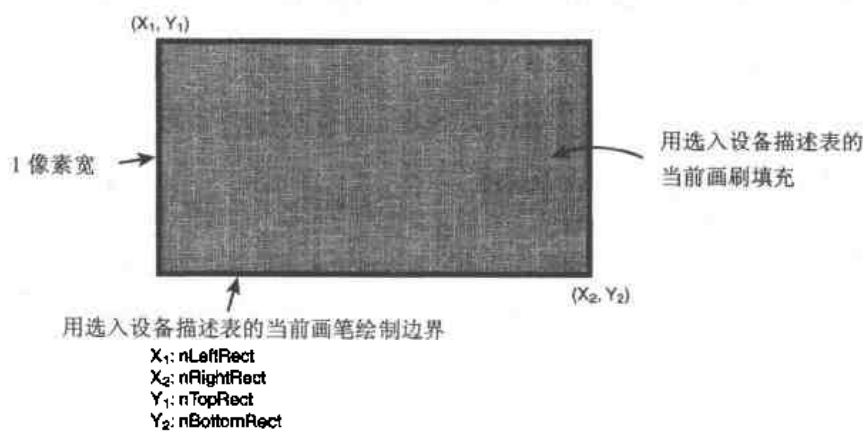


图 4.5 使用 `Rectangle()` 函数

注意



你要注意一个非常重要的细节。传递到 `Rectangle()` 函数的坐标是该矩形的边界框。这就意味着如果线条样式为 `NULL` 的话，将会得到一个实心的矩形，而没有四个边界。

还有两种其他的绘制矩形的更专用的函数：`FillRect()`和 `FrameRect()`，如下所示：

```
int FillRect(HDC hdc, // handle to device context
    CONST RECT *lprc, // pointer to structure with rectangle
    HBRUSH hbr); // handle to brush

int FrameRect(HDC hdc, // handle to device context
    CONST RECT *lprc, // pointer to rectangle coordinates
    HBRUSH hbr); // handle to brush
```

`FillRect()`不使用绘图笔绘制一个填充的矩形，有左上角，但没有右下角。因此，如果要在一个矩形中填充 (10,10) 到 (20,20) 的区域，必须传递矩形结构中的 (10,10) 到 (21,21) 的区域。

而 `FrameRect()`只能绘制有边界的中空的矩形。不可思议的是，`FrameRect()`只使用画刷而不使用画笔。下面是使用 `Rectangle()`函数绘制一个实心填充的矩形的实例：

```
// create the pen and brush
HPEN blue_pen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
HBRUSH red_brush = CreateSolidBrush(RGB(255, 0, 0));
// select the pen and brush into context
SelectObject(blue_pen);
SelectObject(red_brush);

// draw the rectangle
Rectangle(hdc, 10, 10, 20, 20);
// do house keeping...
```

下面是使用 `FillRect()`函数的相似的实例：

```
// define rectangle
RECT rect{10, 10, 20, 20};

// draw rectangle
FillRect(hdc, &rect, CreateSolidBrush(RGB(255, 0, 0)));
```

注意这儿的技巧。和画刷一样，我很匆忙地定义了 `RECT`。画刷根本不需要删除，因为设备描述表中就根本没有选定画刷，这样就很清楚了。

警告



在这些实例中，对 `HDC` 和其他的细节讨论得相当粗略，但我希望你应当了解。很明显，对于任何一个实例，必须都要有一个窗口、`HDC`，每一段都要有合适的开始和结束程序。随着本书内容的进展，我都假设你已经了解了这些问题。

使用 `Rectangle()`函数的实例，可以参见 `DEMO4_3.CPP`；该程序绘制了在窗口表面上

摆动旋转的任意的不同尺寸和颜色的矩形。有一点不同的是所需的句柄是整个窗口的，而不仅仅是用户区的，因此该窗口看上去就像它自己正在被破坏。图 4.6 给出了该程序的输出结果。

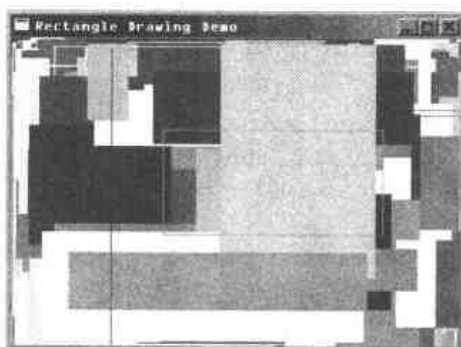


图 4.6 矩形程序 DEMO4_6.CPP

绘制圆

退回到 80 年代，如果那时你能够在计算机上画一个圆，那你简直就是大智者。那时有许多方法来画圆，使用明确的公式：

$$(x-x_0)^2+(y-y_0)^2=r^2$$

或者可以使用 sin 和 cos 函数：

$$x = r\cos(\text{angle})$$

$$y = r\sin(\text{angle})$$

或者使用查找表。关键是在绘制图形的过程中，圆并不是能够最快生成的图形。这个问题以前是个难题，但使用了 500MHz 的 Pentium II 处理器后，就不再是个难题了。GDI 有一个画圆的函数——实际上 GDI 是按照绘制椭圆的方式来画圆。

如果从几何图形上讲，椭圆在任何一个轴上都像是一个东倒西歪的圆。椭圆具有一个长轴和一个短轴，如图所示。中心在 (x_0, y_0) 的椭圆公式如图 4.7 所示。

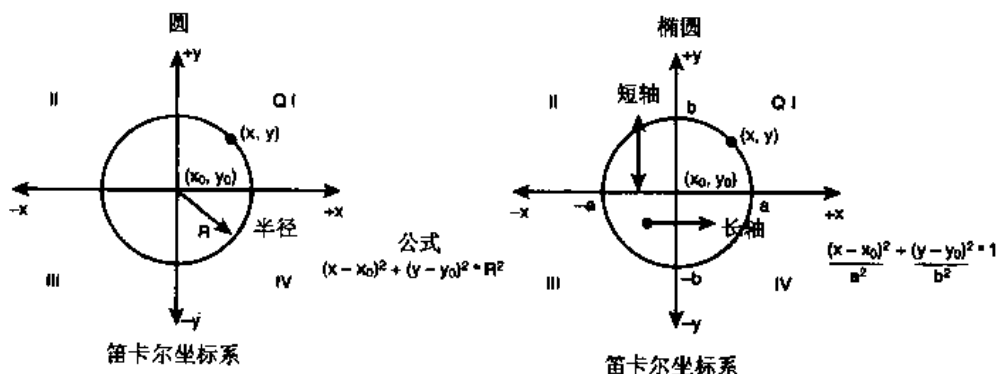


图 4.7 圆和椭圆的数学模型

你可能会认为 GDI 使用了一些相同的概念（定义椭圆的长轴和短轴），但是实际上 GDI 采用了一些不同的方法来定义椭圆。通过 GDI，可以绘制一个有边界的矩形，GDI 则绘制被这个矩形包围的椭圆。本质上讲，定义椭圆的原点的同时也在定义长轴和短轴。

`Ellipse()` 是绘制椭圆的函数，使用当前的画笔和画刷来绘制椭圆。下面是该函数的原型：

```
BOOL Ellipse(HDC hdc,      // handle of device context
             int nLeftRect, // x-coord. of bounding
                        // rectangle's upper-left corner
             int nTopRect,  // y-coord. of bounding
                        // rectangle's upper-left corner
             int nRightRect, // x-coord. of bounding
                        // rectangle's lower-right corner
             int nBottomRect, // y-coord. bounding
                        // rectangle's flower-right corner
```

因此要画圆的话，必须首先确认有边界的矩形是正方形。例如，要绘制一个圆心为（20，20）、半径为 10 的圆，应当：

```
Ellipse(hdc, 10, 10, 30, 30);
```

懂了吗？要绘制一个真正的椭圆，长轴为 100、短轴为 50、原点为（300，200），应当：

```
Ellipse(hdc, 250, 175, 350, 225);
```

CD 上的 DEMO4_4.CPP 和相关的可执行文件显示了一个正在绘制椭圆的实例。该程序通过一个简单的擦除、移动、绘制的动画循环方式绘制了一个正在移动的椭圆。动画循环的类型和我们后面所使用的双缓冲技术或页码倒换技术非常相似，但是使用上述两种技术不可能看到演示程序中的更新过程，因此并没有闪烁！如果有兴趣的话，可以尝试摆弄一下这个演示程序，改变周围的东西，看是否能够领会到如何添加更多的椭圆。

绘制多边形

我们讨论的最后一种基本图形是多边形。目的是迅速绘制开放的或闭合的多边形对象。绘制多边形的程序是 `Polygon()`，如下所示：

```
BOOL Polygon(HDC hdc,      // handle to device context
             CONST POINT *lpPoints, // pointer to polygon's vertices
             int nCount);    // count of polygon's vertices
```

只要向 `Polygon()` 传递一个 `POINT` 的列表及其数量，就可以使用当前的画笔和画刷绘制一个闭合的多边形。图 4.8 给出了这个多边形。

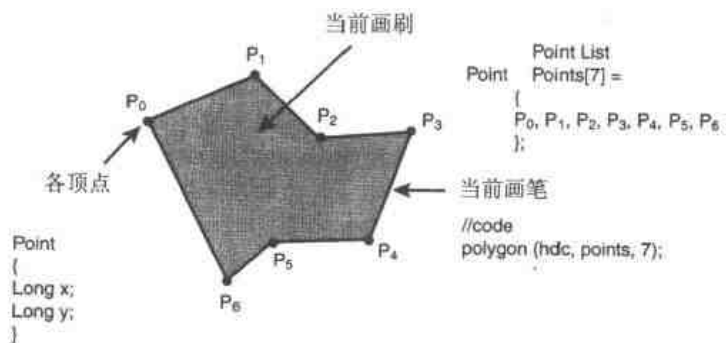


图 4.8 使用 Polygon() 函数

下面是一个实例：

```
// create the polygon show in the figure
POINT poly[7] = {p0x, p0y, p1x, p1y, p2x, p2y,
p3x, p3y, p4x, p4y, p5x, p5y, p6x, p6y, };

// assume hdc is valid, and pen and brush are selected into
// graphics device context
Polygon(hdc, poly, 7);
```

就是这样简单！当然，如果传递绘制一个非凸的多边形或者是自我封闭的多边形的点，GDI 会尽最大可能来绘制，但不能保证绘制得好。

作为一个绘制填充的多边形的实例，DEMO4_5.CPP 绘制了充满整个屏幕的一系列的任意 3~10 点的多边形，每个一次出现的多边形之间稍微有一点延迟，因此可以看到不可思议的结果伴随着非凸的多边形顶点列出时发生。图 4.9 显示了该程序的实际结果。注意由于这些点是任意的，这些多边形由于图形重叠的原因几乎都是非凸的。你能够找到一种方法来确认所有的点都在一个凸包上吗？

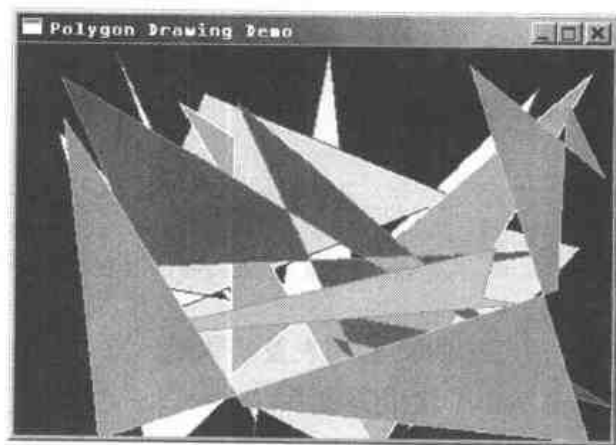


图 4.9 多边形绘制程序 DEMO4_5.CPP 的输出结果

关于文本和字体

使用字体是一个极端复杂的工作，并且我也不想研究实际的东西。如果你想在字体方面撰写一篇有深度的论文的话，最好的方法是查阅一下 Petzold 的《Programming Windows 95》一书。对于 DirectX 环境下的游戏等产品来讲，大多数情况下都使用自己的字体引擎来编写自己的文本。使用 GDI 来绘制文本的惟一的地方，就是在 GDI 进行游戏编程过程中快速绘制分数或者其他简单信息等等。但是最后必须创建自己的字体系统来获得任意的速度。

关于字体稍微复杂一点的内容，至少要讨论一下如何应用 DrawText()和 TextOut()函数来改变字体。通过选定一种新字体对象进入图形设备描述表，和选定新画笔或画刷的过程一样。表 4.1 给出了大量的字体常量，如 SYSTEM_FIXED_FONT，表示等宽字体。等宽表示每个字符具有相同的宽度。同比例字体具有不同的间距。要选定一种新字体进入图形描述表，应当：

```
SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
```

无论通过 DrawText()和 TextOut()函数递交的什么 GDI 文本，都以新字体来绘制。如果希望在字体选择上的功能更强大一些，应当使用表 4.4 中所列的系统内部 TrueType 标准字体。

表 4.4 TrueType 字体字样名

字体的字样字符串	实 例
Courier New	Hello World
Courier New Bold	Hello World
Courier New Italic	<i>Hello World</i>
Courier New Bold Italic	<i>Hello World</i>
Times New Roman	Hello World
Times New Roman Bold	Hello World
Times New Roman Italic	<i>Hello World</i>
Times New Roman Bold Italic	<i>Hello World</i>
Arial	Hello World
Arial Bold	Hello World
Arial Italic	<i>Hello World</i>
Arial Bold Italic	<i>Hello World</i>
Symbol	Χελλο Ωορλδ

要创建其中一种字体，应当使用 `CreateFont()` 函数：

```
HFONT CreateFont( int nHeight,          // logical height of font
                  int nWidth,           // logical average character width
                  int nEscapement,       // angle of escapement
                  int nOrientation,      // base-line orientation angle
                  int fnWeight,          // font weight
                  DWORD fdwItalic,       // italic attribute flag
                  DWORD fdwUnderline,    // underline attribute flag
                  DWORD fdwStrikeOut,    // strikeout attribute flag
                  DWORD fdwCharSet,      // character set identifier
                  DWORD fdwOutputPrecision, // output precision
                  DWORD fdwClipPrecision, // clipping precision
                  DWORD fdwQuality,      // output quality
                  DWORD fdwPitchAndFamily, // pitch and family
                  LPCTSTR lpszFace);     // pointer to typeface name string
                                         // as shown in table 4.4
```

该函数的解释太长了，请参阅 Win32 SDK 帮助来了解其中的细节吧。基本上讲，已经填充了所有的令人讨厌的参数，结果得到一个指向要求字体扫描版本的句柄。然后将该字体选定到设备描述表，一切就都 OK 了。

定时的重要性

我们要讨论的下一个主题是定时。尽管定时看上去并不重要，但对于视频游戏来讲非常关键。如果没有定时和合理的延迟，游戏可能会运行得太快或太慢，动画的错觉就将完全丧失。

在第一章“无尽之旅”中，我曾经提到过大多数游戏运行速度是 30fps（每秒的帧数），但是我没有提到如何控制该定时常数。本部分内容中，将讨论一些跟踪时间的一些技术，以及传递时间相关的消息。本书后面将讨论如何反复地应用这些技术，以保证帧频固定，以及如何添加不能够维持高帧频的慢镜头中的参量动画和物理性质。首先，请看一下 WM_TIMER 消息。

WM_TIMER 消息

PC 机具有内置的非常准确的（微秒级）定时系统，但是由于我们在 Windows 环境下编程，我们自己来摆弄定时系统并不是个好主意。我们应当使用 Windows 内部的定时函数（建立在非常准确的硬件定时器基础上）。这种方法的好处就是 Windows 可以将该定时器虚拟为几乎是无限多个虚拟定时器。因此，为自己着想，尽管在大多数 PC 机上只有一个物理定时器，也应当从大量的定时器消息开始，并且大量接收定时器的消息。

Windows 升级的计时器

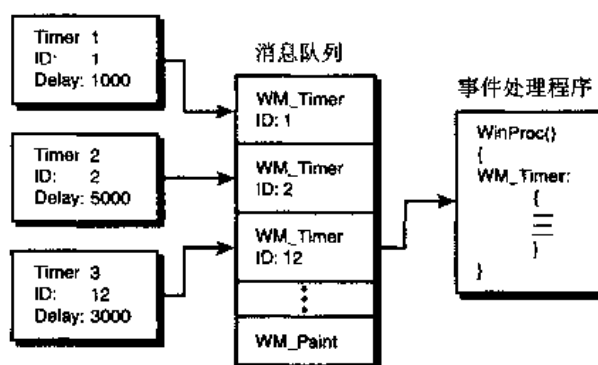


图 4.10 WM_TIMER 消息的消息流

当创建一个定时器时，要自己设定定时器以及延迟的 ID。该定时器将以指定的间隔时间向 WinProc() 传递消息。请看图 4.10 中一些定时器的数据流。每个定时器都经过一定时间后传递 WM_TIMER 消息。在通过创建定时器时设置的定时器 ID 来处理 WM_TIMER 消息时，从其他定时器来通知其中一个定时器。让我们看一看创建定时器的函数 SetTimer()：

```
UNIT SetTimer(HWND hwnd, // handle to parent window
              UINT nIdEvent, // timer id
              UINT nElapse, // time delay in milliseconds
              TIMERPROC lpTimerFunc); // timer callback
```

要创建定时器，还需要：

- 窗口句柄
- 选择标识符
- 微秒级的时间延迟

具有了上述三种内容，才能开始创建定时器。但对于最后一个参数应当稍微解释一下。LpTimerFunc() 和 WinProc() 一样是一个回调函数，因此可以创建一个能够以指定的间隔通过 WM_TIMER 消息调用不在 WinProc() 中处理的函数。该内容已经超出范围，我经常使用 WM_TIMER 消息，并且设定 TIMERPROC 为 NULL。

你可以按需创建多个定时器，但是请记住，定时器也占用资源。如果函数失败将返回 0。否则 SetTimer() 返回创建定时器时使用的定时器标识符。

下一个问题就是如何从其他定时器通知另一个定时器。方法是在传递 WM_TIMER 消息时询问 wParam；wparam 中含有原来创建定时器时的定时器标识符。下面是创建两个定时器的实例，一个是有 1.0 秒延迟，另一个为 3.0 秒延迟：

```
#define TIMER_ID_1SEC 1
#define TIMER_ID_3SEC 2

// maybe do this in WM_CREATE
SetTimer(hwnd, TIMER_ID_1SEC, 1000, NULL);
SetTimer(hwnd, TIMER_ID_3SEC, 3000, NULL);
```

注意，延迟是以毫秒为单位的，1000 毫秒等于 1 秒。下面是添加到 `WinProc()` 函数中用来处理定时器消息的代码：

```
case WM_TIMER:
{
    // what timer fired?
    switch(wparam)
    {
        case TIMER_ID_1SEC:
        {
            // do processing here
        } break;

        case TIMER_ID_3SEC:
        {
            // do processing here
        } break;

        default: break;

    } // end switch

    // let windows know we handled the message
    return(0);

} break;
```

最后，使用定时器结束后，应当使用 `KillTimer()` 删除定时器：

```
BOOL KillTimer(HWND hWnd, // handle of window
               UINT uIDEvent, // timer id
```

继续上面的实例，应当删除在 `WM_DESTROY` 消息中的所有定时器，如下所示：

```
case WM_DESTROY:
{
    // kill timers
    KillTimer(hWnd, TIMER_ID_1SEC);
    KillTimer(hWnd, TIMER_ID_3SEC);

    // terminate application or whatever...
    PostQuitMessage(0);

} break;
```

警告



尽管定时器看上去很自由并且很多，但是 PC 机并不是星际旅行 (Star Trek) 中的计算机。定时器占用资源，应当尽量少用。删除运行时不再需要的定时器。

CD 上的 `DEMO4_6.CPP` 给出了使用定时器的实例。该程序在不同的时间创建了三个定时器，当每个定时器改变时输出结果。最后，尽管定时器采用毫秒级的时间延迟，但是也

很难准确到毫秒级。不要指望你的定时器准确到 10~20 毫秒。要想定时器更准确的话，可以使用 Win32 高性能定时器，或者使用基于 RDTSC 集成语言指令的 Pentium 处理器实时硬件计数器等方法。

低级定时控制

尽管创建定时器是跟踪时间的一种方式，但是该技术也会遇到一些困难：首先，定时器传递消息，第二，定时器并不那么准确，最后，在大多数游戏循环中，都希望能够强制主程序以指定帧频运行，并且帧频不能过高；这可以通过定时程序锁定帧频来达到。定时器在这方面并不擅长。真正需要的是查询系统时钟，然后运行差动测试来检查过去了多少时间。Win32 API 具有这样的功能，称之为 `GetTickCount()`：

```
DWORD GetTickCount(void);
```

`GetTickCount()` 返回从 Windows 启动后的毫秒数。这样做看上去并不能作为一个绝对的参考，因为根本没有绝对的参考，但是可以作为一个很好的差动参考。在定时程序块的前面查询当前的小点计数，在程序循环后面再次查询一次，然后比较二者的差值。这样就可以获得毫秒级的时间差。例如下面是如何确认程序块以准确的 30fps 或 $1/30\text{fps}=33.33$ 毫秒的延迟方式运行：

```
// get the starting time
DWORD start_time = GetTickCount();

// do work, draw frame, whatever

//now wait until 33 milliseconds has elapsed
while ((GetTickCount() - start_time) < 33)
```

上面就是我要讨论的内容。例如，一个正在执行的循环运行 `While()` 逻辑会非常浪费时间，但是可以作为一个分支程序，并且偶尔测试一下。关键是使用这种技术能够在程序块中约束时间。

注意



很明显，如果你的个人电脑不能以 30fps 速度运行，该循环将花费更长的时间。但是如果在你的程序任意运行期间，该循环都以 30~100fps 的速率运行，前面程序就会将程序运行速率锁定 30fps 上，这就是关键。

CD 上的 DEMO4_7.CPP 给出了一个实例，基本上将帧速锁定在 30fps 上，并且在每一个画面上都更新屏幕保护。下面是运行该动作的 `WinMain()` 部分程序代码：

```
// get the dc and hold onto it
hdc = GetDC(hwnd);

// seed random number generator
srand(GetTickCount());
```

```
// endpoints of line
int x1 = rand()%WINDOW_WIDTH;
int y1 = rand()%WINDOW_HEIGHT;
int x2 = rand()%WINDOW_WIDTH;
int y2 = rand()%WINDOW_HEIGHT;

// initial velocity of each end
int x1v = -4 + rand()%8;
int y1v = -4 + rand()%8;
int x2v = -4 + rand()%8;
int y2v = -4 + rand()%8;

// enter main event loop, but this time we use PeekMessage()
// instead of GetMessage() to retrieve messages
while(TRUE)
{
    // get time reference
    DWORD start_time = GetTickCount();

    // test if there is a message in queue, if so get it
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
        // test if this is a quit
        if (msg.message == WM_QUIT)
            break;

        // translate any accelerator keys
        TranslateMessage(&msg);

        // send the message to the window proc
        DispatchMessage(&msg);
    } // end if

    // is it time to change color
    if (++color_change_count >= 100)
    {
        // reset counter
        color_change_count = 0;

        // create a random colored pen
        if (pen)
            DeleteObject(pen);

        // create a new pen
        pen = CreatePen(PS_SOLID,1,
            RGB(rand()%256,rand()%256,rand()%256));

        // select the pen into context
        SelectObject(hdc,pen);
    }
}
```

```
    } // end if

    // move endpoints of line
    x1+=x1v;
    y1+=y1v;

    x2+=x2v;
    y2+=y2v;

    // test if either end hit window edge
    if (x1 < 0 || x1 >= WINDOW_WIDTH)
    {
        // invert velocity
        x1v=-x1v;

        // bum endpoint back
        x1+=x1v;
    } // end if

    if (y1 < 0 || y1 >= WINDOW_HEIGHT)
    {
        // invert velocity
        y1v=-y1v;

        // bum endpoint back
        y1+=y1v;
    } // end if

    // now test second endpoint
    if (x2 < 0 || x2 >= WINDOW_WIDTH)
    {
        // invert velocity
        x2v=-x2v;

        // bum endpoint back
        x2+=x2v;
    } // end if

    if (y2 < 0 || y2 >= WINDOW_HEIGHT)
    {
        // invert velocity
        y2v=-y2v;

        // bum endpoint back
        y2+=y2v;
    } // end if

    // move to end one of line
    MoveToEx(hdc, x1,y1, NULL);
```

```
// draw the line to other end
LineTo(hdc,x2,y2);

// lock time to 30 fps which is approx. 33 milliseconds
while((GetTickCount() - start_time) < 33);

// main game processing goes here
if (KEYDOWN(VK_ESCAPE))
    SendMessage(hwnd, WM_CLOSE, 0,0);

} // end while

// release the device context
ReleaseDC(hwnd,hdc);

// return to Windows like this
return(msg.wParam);

} // end WinMain
```

除了定时程序块以外，还有其他的值得一看的逻辑过程：碰撞逻辑。应当注意到线段有两个端点，每个端点都有其位置和速度。随着线段的移动，程序测试该线段是否和窗口用户区的边缘碰上。如果碰上，该线段从该边缘弹回来，产生一个反弹线的幻觉。

技巧



如果仅仅是延迟你的程序，可以使用 Win32 API 函数 Sleep()。只要将希望延迟的毫秒数的时间延迟传递给该函数，该程序就将执行。例如，要延迟 1.0 秒，应当说 Sleep(1000)。

使用控件

在游戏编程方面的书中我一般不讨论 Windows 控件功能，但是由于可能你需要了解一些 Windows 控件来作为工具，并且我已经收到了很多电子邮件，请求在本书中添加上该部分内容，那现在就讨论一下该方面的内容，仅仅是一点点的内容。

Windows 子控件实际上就是窗口本身。下面是一些最常用的子控件的简单列表：

- 静态文本框
- 编辑框
- 按钮
- 列表框
- 滚动条

另外，还有大量的子按钮类型，如：

- 按钮
- 复选框
- 单选按钮

每一个子按钮类型还有更下一级的子类型。毋庸置疑，即使最复杂的窗口控件也是这些基本类型的混合体。例如一个文件目录控件只是一些列表框、一些文本编辑框和一些按钮。使用这儿所列的基本控件，就可以处理任何事情。一旦精通了其中一方面，其他方面也几乎完全一样，只是一些细节上有所区别，因此我们下面只讨论如何使用包括按钮在内的少数几个子控件。

按钮

Windows 支持大量的按钮类型。如果你正在阅读本书，希望你曾经用过 Windows，并且至少比较熟悉按钮、复选框和单选按钮，这样我就不必事无巨细。我们只讨论如何创建所希望的按钮类型，并且对所传递的消息进行反应。而剩下的内容就需要你自己去考虑了。首先看一下表 4.5，表 4.5 列出了有用的按钮类型。

表 4.5 按钮样式

值	描 述
BS_PUSHBUTTON	创建一个按钮，当用户选定该按钮时向自己的窗口发送一个 WM_COMMAND 消息
BS_RADIOBUTTON	创建一个带文本的小圆圈。默认情况下，该文本在该圆圈的右侧显示
BS_CHECKBOX	创建一个带文本的小的空复选框。默认情况下，该文本在该圆圈的右侧显示
BS_3STATE	创建一个按钮，该按钮和复选框完全相同，不同之处是该复选框在选定或取消选定的情况下都显示灰色
BS_AUTO3STATE	创建一个按钮，该按钮和三态复选框完全相同，不同之处是该框在用户选定时会改变其状态。该状态按照选定、变成灰色和取消选定三种状态循环
BS_AUTOCHECKBOX	创建一个按钮，该按钮和复选框完全相同，不同之处是每次用户选定该复选框时，该复选状态在选定和取消选定之间自动切换
BS_AUTORADIOBUTTON	创建一个按钮，该按钮和单项按钮完全相同，不同之处是当用户选定该复选框时，Windows 自动设定该按钮的复选状态为选定，而设定所有其他同一组的按钮的复选状态为取消选定状态
BS_OWNERDRAW	创建一个属主描述按钮。本窗口在创建该按钮时检索 WM_MEASUREITEM 消息，在改变该按钮的可视方面时检索 WM_DRAWITEM

要创建一个子控件按钮，只要使用“button”作为类字符串和表 4.5 中所列的按钮样式来创建一个窗口即可。然后，当操作该按钮时，它向你的窗口传递 WM_COMMAND 消息，如图 4.11 所示。程序员可以和通常一样处理 wParam 和 lParam，确认子控件传递什么消息，向谁传递。

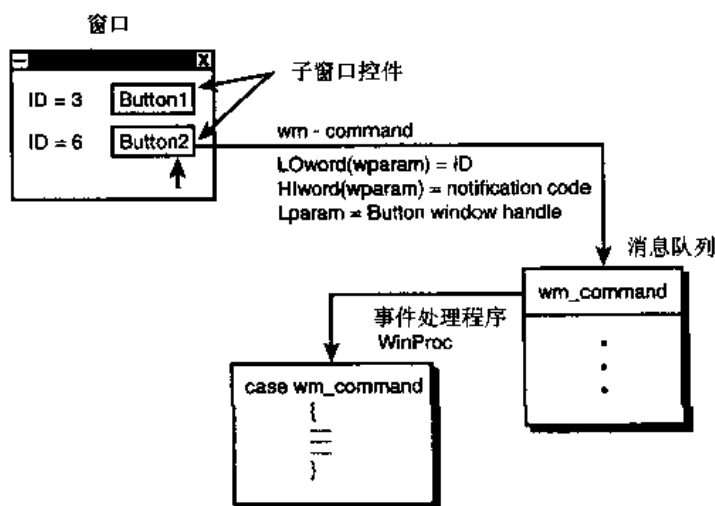


图 4.11 子窗口消息传递

下面看一下需要向创建子按钮控件的 CreateWindowEx() 传递的确切参数。首先，需要设定该类名称为“button”。然后，需要设定样式标志为 WS_CHILD 或者 WS_VISIBLE，并设定表 4.5 中的按钮样式。然后将菜单句柄或 HMENU 放置在正常位置上，发送表示按钮的标识符（当然必须将该标识符分配给 HMENU），创建子按钮控件参数大概就是这些。

下面实例给出了如何创建标识符等于 100、带有“Push Me”的文本的按钮：

```

CreateWindowEx(NULL, // extended style
    "button", // class
    "Push Me", // text on button
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON
    10, 10, // initial x, y
    100, 24, // initial width, height
    main_window_handle, // handle to parent
    (HMENU)(100), // id of button, notice cast to HMENU
    hinstance, // instance of this application
    NULL); // extra creation parms

```

很简单，不是吗？当按下该按钮时，WM_COMMAND 消息将按下面参数进行参数化，并被发送给母窗口的 WinProc()：

```

msg: WM_COMMAND

LOWORD(wparam): Child Window id

```

HIWORD(wparam): Notification Code
 lparam: Child Window Handle

看上去合理吗？惟一的秘密就是通报码。通报码描述的是发生在该按钮控件上的事件，以 BN_ 开头。表 4.6 列出了所有可能的通报码和值。

表 4.6 按钮的通报码

代 码	值
BN_CLICKED	0
BN_PAINT	1
BN_HLITE	2
BN_UNHILITE	3
BN_DISABLE	4
BN_DOUBLECLICKED	5

最重要的通报码当然是 BN_CLICKED 和 BN_DOUBLECLICKED。要处理一个按钮子控件，比如一个简单的按钮，在 WM_COMMAND 事件句柄中按照下面方法操作：

```
// assume a child button was created with id 100
case WM_COMMAND;
{
    // test for id
    if (LOWORD(wparam) == 100)
    {
        // do whatever
    } // end if

    // process all other child controls, menus etc.

    // we handled it
    return(0);

} break;
```

DEMO4_8.CPP 给出了一个实例，创建了一个所有按钮类型的列表，并且显示所有的消息以及在点击和操作按钮时每个消息的 wparam 和 lparam。图 4.12 显示了运行中的该程序。通过该实例，你应该能够更好地理解按钮子控件如何工作。

如果运行 DEMO4_8.EXE，你很快就会发现尽管 WinProc()正在传递描述用户正在处理控件的消息，但是你还是不知道在程序中如何改变或操作该控件。更具体地讲，一些控件在点击时似乎并没有执行什么操作。这是非常重要的问题，因此下面简单讨论一下。

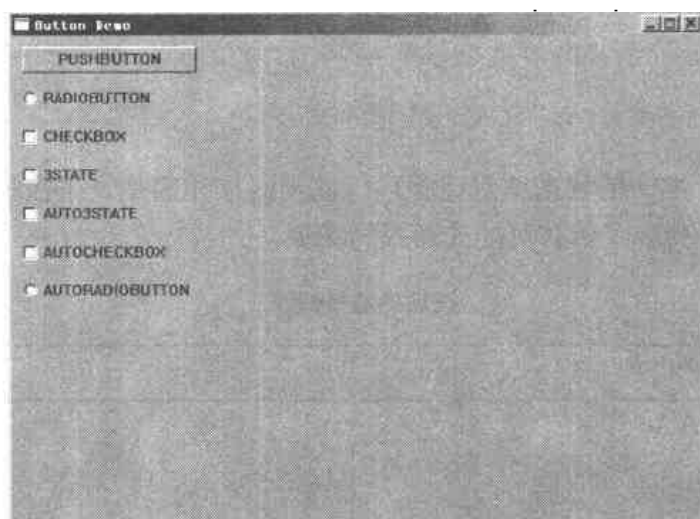


图 4.12 DEM04_8.EXE 子控件程序

将消息传递到子控件

由于子控件就是窗口，因此它们和其他窗口一样接收消息。但由于它们是母窗口的子控件，就 WM_COMMAND 消息而言，该消息被传递到母窗口。然而，向子控件（如按钮）传递消息也是可能的，子控件使用自己的默认的 WinProc() 处理该消息。这就是如何改变控件状态的方法——即向控件传递消息。

就按钮而言，有大量的消息可以传递到按钮控件，使用 SendMessage() 来改变按钮的状态，并（或）检索该按钮的状态。请记住，SendMessage() 同样返回一个值。下面是一些有用的消息列表，以及 wParam 和 lParam 的参数。

目的：模拟点击按钮。

```
msg:          BM_CLICK
wparam:       0
lparam:       0
```

例如：

```
// this would make the button look like it was pressed
SendMessage(hwndbutton, BM_CLICK, 0, 0);
```

目的：用于在复选框中或单选按钮中设置为选定。

```
msg:          BM_SETCHECK
wparam:       fCheck
lparam:       0
```

fCheck 可以是下面其中一项:

值	描 述
BST_CHECKED	设定按钮状态为选定
BST_INDETERMINATE	设定按钮状态为灰色, 表示未确定的状态 只有在该按钮为 BS_3STATE 或 BS_AUTO3STATE 样式时使用该值
BST_UNCHECKED	设定按钮状态为取消选定

例如:

```
// this would check a check button
SendMessage(hwndbutton, BM_SETCHECK, BST_CHECKED, 0);
```

目的: 用来检索该按钮选定的状态。可能返回如下值:

```
msg:          BM_GETCHECK
wparam:       0
lparam:       0
```

值	描 述
BST_CHECKED	选定该按钮
BST_INDETERMINATE	设定按钮状态为灰色, 表示未确定的状态 只有在该按钮为 BS_3STATE 或 BS_AUTO3STATE 样式时使用该值
BST_UNCHECKED	取消选定该按钮

例如:

```
// this would get the check state of a checkbox
if (SendMessage(hwndbutton, BM_GETCHECK, 0, 0) == BST_CHECKED)
{
    // button is checked
} //end if
else
{
    // button is not checked
} // end else
```

目的: 用来突出显示用户选定的按钮。

```
msg:          BM_SETSTATE
wparam:       fState
lparam:       0
```

突出显示该按钮时 `fState` 为真，否则为假。

例如：

```
// this would highlight the button control
SendMessage(hwndbutton, BM_SETSTATE, 1, 0);
```

目的：获取该按钮控件的常规状态。可能返回下面的值：

```
msg:          BM_GETSTATE
wparam:       0
lparam:       0
```

值	描 述
BST_CHECKED	指示该按钮被选定
BST_FOCUS	指示聚焦状态。非零值表示该按钮使用键盘
BST_INDETERMINATE	表示设定按钮状态为灰色，因为该按钮为未确定的状态。只有在该按钮为 BS_3STATE 或 BS_AUTO3STATE 样式时使用该值
BST_PUSHED	指定突出显示状态。非零值表示该按钮突出显示。当用户将光标定位在一个按钮上或者在该按钮上按住鼠标左键时，自动突出显示该按钮。当用户释放鼠标按钮时取消突出显示
BST_UNCHECKED	表示取消选定该按钮

例如：

```
// this code can be used to get the state of the button
switch(SendMessage(hwndbutton, BM_GETSTATE, 0, 0))
{
// what is the button state
case BST_CHECKED:      { } break;
case BST_FOCUS:        { } break;
case BST_INDETERMINATE: { } break;
case BST_PUSHED:       { } break;
case BST_UNCHECKED:    { } break;

default: break;
} // end switch
```

上述内容就是由于子控件的内容。现在你至少已经了解了子控件是什么以及如何处理子控件等问题。下面应当是从 Windows 查询信息的内容了。

获取信息

华尔街的 Gordon Gecko 曾经说过：“为什么不停止向我发送信息并从我这儿获取信息呢？”这句话对于当前情况以及许多其他事情上都非常合适。系统的信息对于游戏尽可能地利用一个系统所能提供的资源是至关重要的。的确，Windows 包含有大量的信息检索函数，能够获得关于 Windows 设置及其硬件本身的众多的细节。

Win32 支持大量的 Get*()函数，而 DirectX 支持大量的 GetCaos*()函数。我们只讨论一些经常使用的 Win32 函数。在本书的下一部分中将学习更多的 DirectX 支持的信息检索函数。那些函数主要和光谱多媒体终端有关。

下面介绍一下我经常使用的三个函数。实际上，可以通过一个 Get 类函数来查询关于 Windows 系统的所有知识。只要在编译器帮助中将“get”输入 Win32 SDK 搜索引擎，就可以得到所有需要的内容。我们了解一下如何使用这三个函数。

第一个函数是 GetSystemInfo()。主要返回关于处理正在使用的硬件的信息，如处理器类型、有多少个处理器等内容。下面是该函数的原型：

```
VOID GetSystemInfo(
    LPSYSTEM_INFO lpSystemInfo);
// address of system information structure
```

该函数值采用一个指向 SYSTEM_INFO 结构的指针，并且填充所有的字段。下面是 SYSTEM_INFO 结构的内容：

```
typedef struct _SYSTEM_INFO
{ // sinf
    union {
        DWORD dwOemId;
        Struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO;
```

有关这些字段的具体细节非常多，我们对此不进行详细的讨论，但是很明显其中有一

些很有趣的字段。例如，`dwNumberOfProcessors` 表示 PC 机主板上处理器的数目。`DwProcessortype` 表示处理器的实际类型，可以采用下面的值：

值
<code>PROCESSOR_INTEL_386</code>
<code>PROCESSOR_INTEL_486</code>
<code>PROCESSOR_INTEL_PENTIUM</code>

其他字段都是不需加以说明的，具体内容请看 Win32 SDK。如果认真考虑该函数的话，它是一个令人惊奇的函数。你能设想确定安装的处理器类型是多么困难的事吗？更不必说有多少个处理器，以及从什么地方开始等等问题。

首先编写一个非常复杂的可以探测 486、Pentium、Pentium II 处理器等等的检测算法，通过读写操作来探测消息，直到检测到计算机上的处理器类型为止。当然，Pentium 类的处理器具有标识符字符串和计算机标志，但是 486 处理器就很难确定了。关键是获取系统内部信息需要一个很复杂的函数。

下面是一个非常通用的能够检索 Windows 和 Desktop 的所有信息的函数，`GetSystemMetrics()`：

```
int GetSystemMetrics(int nIndex); // system metric or configuration setting
to retrieve
```

`GetSystemMetrics()` 功能非常强大。只要将检索数据的索引传递给该函数，如表 4.7 所示，就会返回所需要的信息。另外，表 4.7 是本书中最长的表格。因为我不喜欢察看帮助，因此就将下面内容从帮助中截取下来，以利于方便地使用。

表 4.7 `GetSystemMetrics()` 的系统度量常数

值	描 述
<code>SM_ARRANGE</code>	指定系统安排最小化窗口方法的标志，对于大多数最小化窗口的信息来讲，请看后面的说明
<code>SM_CLEANBOOT</code>	指定系统如何启动的值： 0 正常启动 1 安全模式启动 2 网络故障安全模式
<code>SM_CMOUSEBUTTONS</code>	鼠标按钮的数目，如果没有安装鼠标，则为 0
<code>SM_CXBORDER, SM_CYBORDER</code>	窗口边界的宽度和高度（以像素表示）。和具有三维视角的窗口的 <code>SM_CXEDGE</code> 相同
<code>SM_CXCURSOR, SM_CYCURSOR</code>	光标的宽度和高度（以像素表示），是当前显示驱动程序支持的光标尺寸。系统不能创建其他尺寸的光标

续表

值	描 述
SM_CXDOUBLECLK, SM_CYDOUBLECLK	双击操作中第一次点击位置周围矩形的宽度和高度（以像素表示），第二次点击必须在此矩形范围内，以便于系统能够确认两次点击是双击（这两次点击必须在指定时间内完成）
SM_CXDRAG, SM_CYDRAG	以拖动点为中心的、在拖动操作开始之前允许鼠标指针有限移动的矩形的宽度和高度（以像素表示）。这样就可以使用户能够容易点击和释放鼠标按钮，而不会无意识地开始拖动操作
SM_CXEDGE, SM_CYEDGE	一个 3D 边界的尺寸（以像素表示），是 SM_CXBORDER 和 SM_CYBORDER 的相应的 3D 版本
SM_CXFIXEDFRAME, SM_CYFIXEDFRAME	一个有标题而无大小的窗口的边界线宽度（以像素表示）。SM_CXFIXEDFRAME 是其水平边界线的宽度，SM_CYFIXEDFRAME 是其垂直边界线的宽度
SM_CXFULLSCREEN, SM_CYFULLSCREEN	全屏窗口中用户区的宽度和高度。要获得没有被用户区盖住的部分屏幕的坐标，调用带 SPI_GETWORKAREA 值的 SystemParameterInfo 函数
SM_CXHSCROLL, SM_CYHSCROLL	指向水平滚动条的箭头位图的宽度（以像素表示），以及水平滚动条的高度（以像素表示）
SM_CXHTHUMB	水平滚动条中按钮框的宽度（以像素表示）
SM_CXICON, SM_CYICON	图表的默认宽度和高度（以像素表示）。这些值一般都是 32×32，但也可以根据安装的显卡的类型而不同
SM_CXICONSPACING, SM_CYICONSPACING	以大图标样式表示的项的网格单元的尺寸（以像素表示）。图表尺寸调整时，每个项都和该尺寸的矩形相适应。这些值一般都大于或等于 SM_CXICON 和 SM_CYICON 的值
SM_CXMAXIMIZED, SM_CYMAXIMIZED	一个最大化的上层窗口的默认尺寸（以像素表示）
SM_CXMAXTRACK, SM_CYMAXTRACK	一个有标题和边界尺寸的窗口的默认尺寸（以像素表示）。用户不能拖动该窗口大于该尺寸。可以通过处理 WM_GETMINMAXINFO 消息来覆盖该窗口的尺寸
SM_CXMENUCHECK, SM_CYMENUCHECK	默认菜单复选标志位图的尺寸（以像素表示）
SM_CXMENUSIZE, SM_CYMENUSIZE	菜单栏按钮（如多重文件的关闭子菜单）的尺寸（以像素表示）
SM_CXMIN, SM_CYMIN	窗口的最小宽度和高度（以像素表示）

续表

值	描 述
SM_CXMINIMIZED, SM_CYMINIMIZED	一个常规最小化窗口的尺寸 (以像素表示)
SM_CXMINSIZING, SM_CYMINSIZING	最小化窗口网格单元的尺寸 (以像素表示)。当重新调整尺寸时, 每个最小化窗口都和该尺寸矩形相适应。这些值一般都大于或等于 SM_CXMINIMIZED 和 SM_CYMINIMIZED 的值
SM_CXMINTRACK, SM_CYMINTRACK	窗口最小化跟踪宽度和高度 (以像素表示)。用户不能拖动该窗口框小于该尺寸。可以通过处理 WM_GETMINMAXINFO 消息来覆盖该窗口的尺寸
SM_CXSCREEN, SM_CYSCREEN	屏幕的宽度和高度 (以像素表示)
SM_CXSIZE, SM_CYSIZE	窗口标题栏中按钮的宽度和高度 (以像素表示)
SM_CXSIZEFRAME, SM_CYSIZEFRAME	能够改变尺寸的窗口边框的宽度 (以像素表示)。SM_CXSIZEFRAME 是水平边界的宽度, SM_CYSIZEFRAME 是垂直边界的高度
SM_CXSMICON, SM_CYSMICON	小图标的建议尺寸 (以像素表示)。小图标一般以小图标样式显示于窗口标题中
SM_CXSMSIZE, SM_CYSMSIZE	小标题按钮的尺寸 (以像素表示)
SM_CXVSCROLL, SM_CYVSCROLL	垂直滚动条的宽度 (以像素表示), 以及垂直滚动条中箭头位图的高度 (以像素表示)
SM_CYCAPTION	常规标题区域的高度 (以像素表示)
SM_CYKANJIWINDOW	用于窗口双字节字符设置版本, 屏幕底部汉字窗口的高度 (以像素表示)
SM_CYMENU	单行菜单栏的高度 (以像素表示)
SM_CYSMCAPTION	小标题的高度 (以像素表示)
SM_CYVTHUMB	垂直滚动条中按钮框的高度 (以像素表示)
SM_DBCSENABLED	如果安装 USER.EXE 的双字节字符设置版本为真或非零; 否则为假或零
SM_DEBUG	如果安装 USER.EXE 的测试版为真或非零; 否则为假或零
SM_MENUDROPALIGNMENT	如果下拉菜单中相应菜单栏中的项是右对齐的则为真或非零; 否则为假或零
SM_MIDEASTENABLED	如果系统能够使用 Hebrew/Arabic 语言则为真
SM_MOUSEPRESENT	如果安装了鼠标则为真或非零; 否则为假或零
SM_MOUSEWHEELPRESENT	仅用于 Windows NT。如果安装了带轮的鼠标则为真或非零; 否则为假或零

续表

值	描 述
SM_NETWORK	如果存在网络设置就设定最小的有效位，否则就清除有效位。保存其他位以便于将来使用
SM_PENWINDOWS	如果安装了用于画笔计算扩展版本的 Microsoft Windows 则为真或非零；否则为假或零
SM_SECURE	如果应用安全设置则为真；否则为假
SM_SHOWSOUNDS	如果用户要求在只能以听觉的方式显示信息的情况下应用程序显示信息则为真或非零；否则为假或零
SM_SLOWMACHINE	如果计算机有低端（慢）处理器则为真；否则为假
SM_SWAPBUTTON	如果鼠标左键和右键功能交换则为真或非零；否则为假或零

表 4.7 中没有的系统度量常数，就不必去了解了。下面的实例是创建一个和屏幕显示区同样大小的窗口：

```
// create the window
if (!(hwnd = CreateWindowEx(NULL, // extended style
    WINDOW_CLASS_NAME, // class
    "Button Deme", // title
    WS_POPUP | WS_VISIBLE,
    0, 0, // initial x, y
    GetSystemMetrics(SM_CXSCREEN), // initial width
    GetSystemMetrics(SM_CYSCREEN), // initial height
    NULL, // handle to parent
    NULL, // handle to menu
    hinstance, // instance of this application
    NULL))) // extra creation parms
    Return(0);
```

注 意

请注意该实例中使用了 WS_POPUP 窗口样式，而不是 WM_OVERLAPPEDWINDOW。该实例创建了一个没有任何边界或控件的窗口，导致屏幕空白，这种效果可以应用到全屏游戏应用程序中。

作为另一个例子，你可以使用下面的代码来测试鼠标：

```
if (GetSystemMetrics(SM_MOUSEPRESENT))
{
    // there's a mouse
} // end if
else
{
```

```
// no mouse
} // end else
```

最后，当绘制文本时，应当了解 GDI 使用的字体，例如，每个字符的宽度以及其他相关尺寸。如果编写绘制文本的程序代码，并且了解了字体，可以更准确地定位该文本。检索文本规格的函数是 `GetTextMetrics()`：

```
BOOL GetTextMetrics(HDC hdc, // handle of device context
    LPTEXTMETRIC lptm); // address of text metrics structure
```

你可能会对为什么要使用 `hdc` 感到疑惑不解，这是因为可能会有多个被选用的不同字体的设备描述表，因此必须通知上述函数以确定文本的规格。有点小聪明的函数！另外 `lptm` 是一个带有该文本信息的指向 `TEXTMETRIC` 结构的指针，如下所示：

```
typedef struct tagTEXTMETRIC{
    LONG tmHeight;           // the height of the font
    LONG tmAscent;           // the ascent of the font
    LONG tmDescent;          // the descent of the font
    LONG tmInternalLeading;   // the internal leading
    LONG tmExternalLeading;   // the external leading
    LONG tmAveCharWidth;     // the average width
    LONG tmMaxCharWidth;     // the maximum width
    LONG tmWeight;           // the weight of the font
    LONG tmOverhang;         // the overhang of the font
    LONG tmDigitizeAspectX;   // the designed for x-aspect
    LONG tmDigitizeAspectY;   // the designed for y-aspect
    BCHAR tmFirstChar;       // first character font defines
    BCHAR tmLastChar;        // last character font defines
    BCHAR tmDefaultChar;     // char used when desired not in set
    BCHAR tmBreakChar;       // the break character
    BYTE tmItalic;           // is this an italic font
    BYTE tmUnderlined;       // is this an underlined font
    BYTE tmStruckOut;        // is this a strikeout font
    BYTE tmPitchAndFamily;    // family and tech, truetype..
    BYTE tmCharSet;          // what is the character set
} TEXTMETRIC;
```

因为大部分人生活中不常使用打印设备，所以许多字段都是毫无意义的，我用黑体显示了有意义的一些字段。请看下面的术语列表，参考图 4.13，可能对理解这些术语有些帮助。

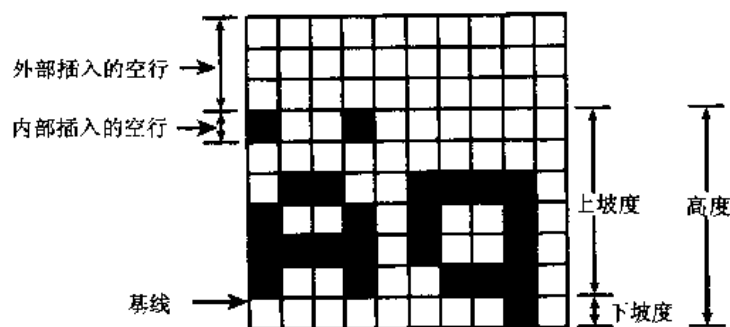


图 4.13 一个字符的构成

高度——字符的总高度（像素表示）。

基线——一个参考点，通常是大写字母的底线。

上坡度——一个重音符号从基线到顶端的像素数。

下坡度——一个小写字母扩展从基线到底边的像素数。

内部插入的空行——重音标记允许用的像素数。

外部插入的空行——在该字符上面允许插入的其他字符的像素数，因此这些字符不能在彼此之间上面运行。

下面是一个如何使文本居中的实例：

```
TEXTMETRIC tm; // holds the textmetric data

// get the textmetrics
GetTextMetrics(hdc, &tm);

// used tm data to center a string given the horizontal width
// assume width of window id WINDOW_WIDTH
int x_pos = WINDOW_WIDTH
strlen("Center This String") *tm.tmAveCharWidth/2;

// print the text at the centered position
TextOut(hdc, x_pos, 0, "Center This String",
        strlen("Center This String"));
```

无论该字体的尺寸为多少，该程序都能令它居中。

T3D 游戏控制程序

在本书开始部分，我曾经提到过如果创建了一个 Windows 外壳应用程序，并且创建一个代码结构，隐藏了运行中的、单调的 Windows 细节，Win32/DirectX 编程几乎就是类似

于 32 位 DOS 的编程过程。现在你已经对此了解得足够多了。本部分内容中，我们将讨论如何装配 T3D 游戏控制程序，从现在开始，该控制程序将是所有演示程序和游戏的基础。

目前，你应当知道要创建一个 Windows 应用程序，需要使用 `WinProc()` 和 `WinMain()` 函数。下面我们创建一个最小的包含这些组件并带有窗口的 Windows 应用程序。该应用程序将调用三个函数，来运行游戏逻辑过程。Windows 消息的处理及其 Win32 相关的语法的细节就不成问题了（除非你希望做）。图 4.14 显示了 T3D 游戏控制程序的结构。

图中可以看到，只需要将三个函数添加到控制程序中：

```
int Game_Init(void *parms = NULL, int num_parms = 0);
int Game_Shutdown(void *parms = NULL, int num_parms = 0);
int Game_Main(void *parms = NULL, int num_parms = 0);
```

`Game_Init()` 在进入 `WinMain()` 中的主事件循环之前调用，并且只调用一次。然后就是初始化游戏中所有内容的步骤了。

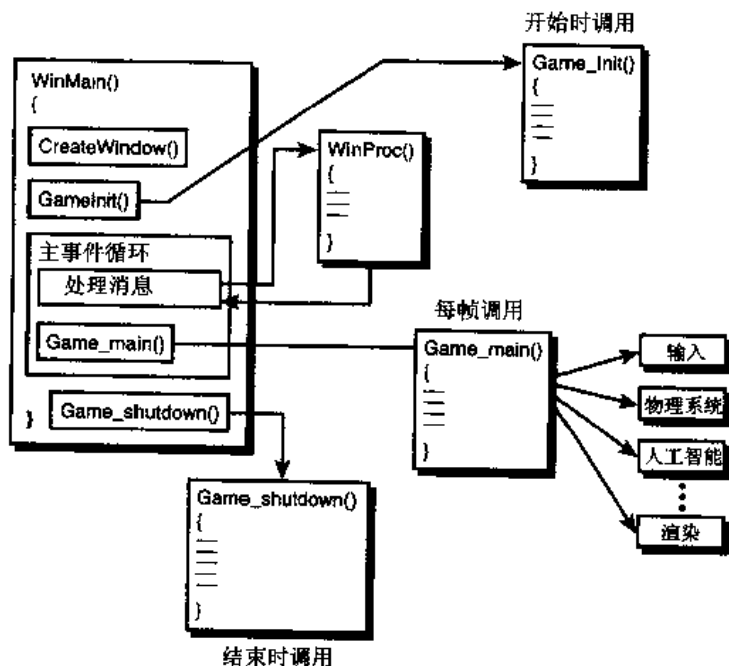


图 4.14 T3D 游戏控制程序的结构

`Game_Main()` 和标准 C/C++ 程序中的 `Main()` 函数相类似，只是它的主事件循环每次处理 Windows 消息之后就调用该函数一次。这也是游戏的整个逻辑过程。你应当处理 `Game_Main()` 中的所有着色、声音、人工智能等内容，或者在 `Game_Main()` 函数中调用相应函数处理。关于 `Game_Main()` 的唯一的警告是你必须绘制一帧画面，然后就返回，因此不能缺少 `WinMain()` 事件处理程序。并且，请记住每次进入和终止该函数，自动变量是瞬间变化的，如果想一直使用该数据，应当将该变量在 `Game_Main()` 中设置为全局变量或局部静态变量。

`Game_Shutdown()` 在 `WinMain()` 中的主事件循环结束后被调用，通过用户发出的消息来

触发该函数，最后发送一个 `Wm_Quit` 消息。在 `Game_Shutdown()` 函数中，应当将所有任务都完成，并清除在玩游戏过程中使用的资源。

文件 `T3DCONSOLE.CPP` 中含有 `T3D` 游戏控制程序，下面是该程序的 `WinMain()` 部分，显示调用所有的控制程序的函数：

```
// WINMAIN //////////////////////////////////////
int WINAPI WinMain(HINSTANCE hinstance,
                  HINSTANCE hprevinstance,
                  LPSTR lpCmdline,
                  int nCmdShow)
{
    WNDCLASSEX winclass; // this holds the class we create
    HWND        hwnd;    // generic window handle
    MSG         msg;      // generic message
    HDC         hdc;      // graphics device context

    // first fill in the window class structure
    winclass.cbSize      = sizeof(WNDCLASSEX);
    winclass.style       = CS_DBLCLKS | CS_OWNDC |
                          CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc = WindowProc;
    winclass.cbClsExtra  = 0;
    winclass.cbWndExtra  = 0;
    winclass.hInstance  = hinstance;
    winclass.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor     = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;
    winclass.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);

    // save hinstance in global
    hinstance_app = hinstance;

    // register the window class
    if (!RegisterClassEx(&winclass))
        return(0);

    // create the window
    if (!(hwnd = CreateWindowEx(NULL, // extended style
                              WINDOW_CLASS_NAME, // class
                              "T3D Game Console Version 1.0", // title
                              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                              0,0, // initial x,y
                              400,300, // initial width, height
                              NULL, // handle to parent
                              NULL, // handle to menu
```

```

        hinstance, // instance of this application
        NULL))) // extra creation parms
return(0);

// save main window handle
main_window_handle = hwnd;

// initialize game here
Game_Init();

// enter main event loop
while(TRUE)
{
    // test if there is a message in queue, if so get it
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // test if this is a quit
        if (msg.message == WM_QUIT)
            break;

        // translate any accelerator keys
        TranslateMessage(&msg);

        // send the message to the window proc
        DispatchMessage(&msg);
    } // end if

    // main game processing goes here
    Game_Main();

} // end while

// closedown game here
Game_Shutdown();

// return to Windows like this
return(msg.wParam);

} // end WinMain

```

稍息片刻，回顾一下 WinMain()函数。该程序看上去非常普通，因为它只是我们一直在使用的众多函数中的一个！惟一的不同是调用 Game_Init()、Game_Main()、Game_Shutdown()函数，如下所示：

```

////////////////////////////////////
int Game_Main(void *parms = NULL)

```

```

{
// this is the main loop of the game, do all your processing
// here

// for now test if user is hitting ESC and send WM_CLOSE
if (KEYDOWN(VK_ESCAPE))
    SendMessage(main_window_handle, WM_CLOSE, 0, 0);

// return success or failure or your own return code here
return(1);

} // end Game_Main

////////////////////////////////////////

int Game_Init(void *parms = NULL)
{
// this is called once after the initial window is created and
// before the main event loop is entered; do all your initialization
// here

// return success or failure or your own return code here
return(1);

} // end Game_Init

////////////////////////////////////////

int Game_Shutdown(void *parms = NULL)
{
// this is called after the game is exited and the main event
// loop while is exited; do all you cleanup and shutdown here

// return success or failure or your own return code here
return(1);

} // end Game_Shutdown

```

该控制台函数的功能就是这样！剩下的就是每次将这些函数添加到游戏程序代码中的工作了。但是我还在 `Game_Main()` 函数中加入了一点东西，测试 Esc 键，传递 `WM_CLOSE` 消息来关闭窗口。这样就可以不必使用鼠标或 `Alt+F4` 组合键来关闭窗口。并且我也相信你注意到了每一个函数参数列表都和下面相似：

```
Game_*(void *parms = NULL, int num_parms = 0);
```

`num_parms` 只是便于程序员向每个函数中传递参数和参数的数量。类型为 `void`，因此使用非常灵活。但这也并不是固定不变的，可以改变它，但是在开始时有点问题。

最后，可能你认为我本应当通过使用 `WS_POPUP` 样式而强制得到全屏显示的不带任何控件的窗口。我会探讨这个问题的，但是我认为在大量的演示程序中将它们设置为有限尺寸的窗口显示更好一些，这样更容易调试。我们也能够在逐个演示程序的基础上改变该样式为全屏显示，因此现在将该演示设置为窗口显示。

C++

如果你是一个 C 程序员，语法 `Game_*(void *parms = NULL, int num_parms = 0)` 看上去就有点不同。默认的赋值称为默认参数。如果已经知道参数和默认值相同的话，只要将这些参数赋值为列出的默认值，而不必输入这些参数。例如，如果不喜欢使用该参数列表，并且也不关心是否 `*parms = NULL` 以及 `int num_parms = 0`，可以调用无参数的 `Game_Main()`。另一方面，如果想传递参数，应当使用 `Game_Main(&list, 12)`，或者相似的设置。如果仍然感到有点模糊的话，请查阅附录 D，简要复习一下 C++。

如果在 CD 上运行 `T3DCONSOLE.EXE`，只能看到一个空白的窗口。但是你只要将 `Game_Init()`、`Game_Main()`、`Game_Shutdown()` 函数以及三维游戏程序添加进去，就可以获得百万美元。当然，我们还有其他的一些方式来完成，但暂时要就此告一段落。

作为最后一次使用 T3D 游戏控制程序的演示实例，我已经创建了在此基础上的应用程序 `DEMO4_9.CPP`。这是一个 3D 的星际世界演示程序，对于使用 GDI 是个不错的实例。检查该程序，看是否能够提高或减缓速度。该程序在此演示了擦除、移动、绘制动画的循环。该程序也通过使用定时程序将帧速限定为 30fps。

总 结

好，亲爱的朋友们，你现在已经是一个 Windows 专家了，至少足以能够应付游戏编程的黑暗帝国了。在本章中，我们讨论了大量的内容，包括图形设备接口、控制、定时、获取信息等等内容。最后，讨论了真实的模板应用程序——T3D 游戏控制程序。使用 T3D 游戏控制程序，可以开始编写一些真正的 Windows 应用程序。从下一章开始，我们将进行 DirectX 的精彩内容的学习，下面内容将更加引人入胜。

第二部分

DirectX 和 2D 基础

第五章

DirectX 基础和令人生畏的 COM

第六章

首次接触：*DirectDraw*

第七章

高级 *DirectDraw* 和位图图形

第八章

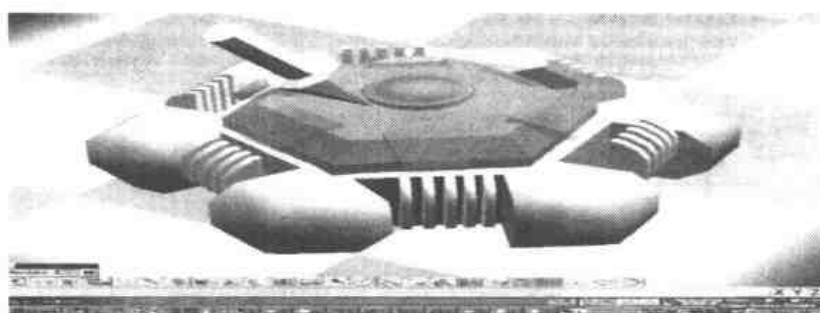
矢量光栅化及 2D 变换

第九章

用 *DirectInput* 和力反馈进行输入

第十章

用 *DirectSound* 和 *DirectMusic* 演奏乐曲



5

DirectX 基础和令人生畏的 COM

本章中，我们将深入学习 DirectX 以及构成这项先进技术的所有基础组件。此外，我们还将详细讨论一下组件对象模型（COM），所有的 DirectX 组件都是由 COM 构成的。如果你只是一个 C 程序员的话，那你应当十分关注这部分内容。不要担心，我们将详细地讨论该内容。

对于该部分内容，提出一点警告：在确定你不想掌握这部分内容之前，请阅读本章全部内容。DirectX 和 COM 是相互关联的，没有另外一个很难解释其中任意一部分内容。例如，如果不定义零，就无法解释零的概念。如果认为 DirectX 和 COM 的关系很简单，那你就完全错误了。

下面是我们要讨论的主要内容：

- ➔ DirectX 介绍
- ➔ 组件对象模型（COM）
- ➔ COM 执行的工作实例
- ➔ DirectX 和 COM 如何协调工作
- ➔ COM 的应用前景

DirectX 基础

我现在感觉自己就像是 Microsoft 的传道士（提醒 Microsoft，请付我薪水），不停地将我的朋友们推向黑暗一边，但是 Microsoft 这个坏家伙一直有更好的技术！我说得对吗？

对程序员而言，DirectX 带有更多的控制功能。但实际上，这样做是值得的。基本上来

讲，DirectX 是从视频、音频、输入、网络以及安装等抽象而来的软件系统，因此无论一台 PC 计算机是怎样的设置，都可以使用相同的程序。另外 DirectX 技术要比 Windows 系统自带的 GDI 和/或 MCI（媒体控制接口）速度快许多倍，功能也更强大。

图 5.1 图示了使用和不使用 DirectX 技术的情况下制作 Windows 游戏的过程。注意 DirectX 方法是多么的清晰雅致。

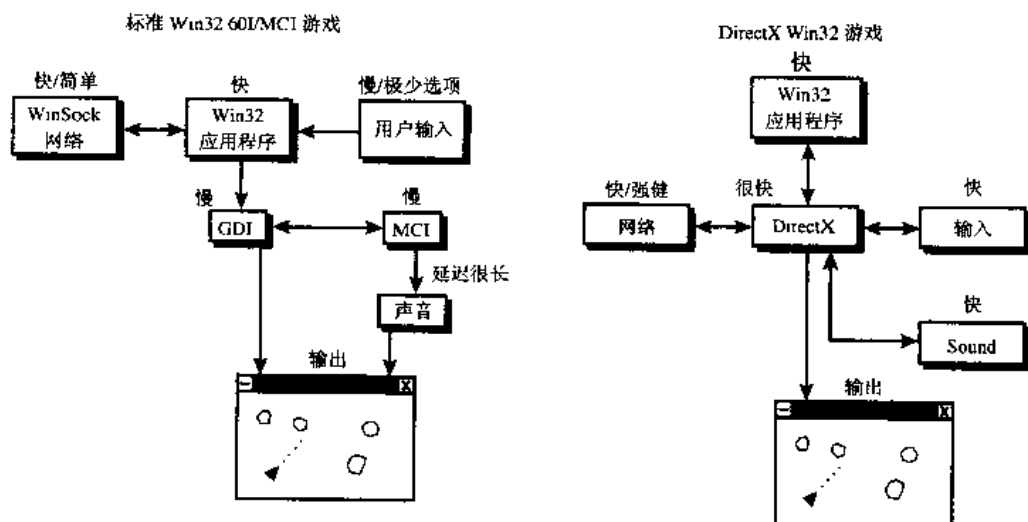


图 5.1 DirectX 和 GDI/MCI

DirectX 的工作原理是什么呢？DirectX 提供了几乎是硬件级控制所有设备的功能。这样就可以通过组件对象模型（COM）技术和由 Microsoft 和硬件厂商编写的一套驱动程序和库函数来完成。Microsoft 提出了一套关于函数、变量、数据结构等内容的协议——硬件厂商也必须遵循以实现驱动程序与和硬件间的交流。

只要遵循这些约定，就不需要担心硬件的细节问题。只要将该内容调用到 DirectX 中，DirectX 就会完成处理细节的工作。无论视频卡、音频卡、输入设备、网卡或是其他设备是什么类型，只要是 DirectX 支持的类型，即使不熟悉这些设备，在程序中也可以使用该设备。

目前有大量的 DirectX 组件，如下所示，图 5.2 给出了其示意图。

- DirectDraw
- DirectSound
- DirectSound3D
- DirectMusic
- DirectInput
- DirectPlay
- DirectSetup
- Direct3DRW

- Direct3DIM

HEL 和 HAL

在图 5.2 中，可以看到在 DirectX 下有两个层面，分别是 HEL（硬件模拟层）和 HAL（硬件抽象层）。这是一个约定：DirectX 是一种非常具有远见的设计思路，它假定可以通过硬件来实现高级性能。但是，如果硬件并不支持某些性能，那怎么办呢？这就是 HAL 和 HEL 双重模式设计思路的基础。

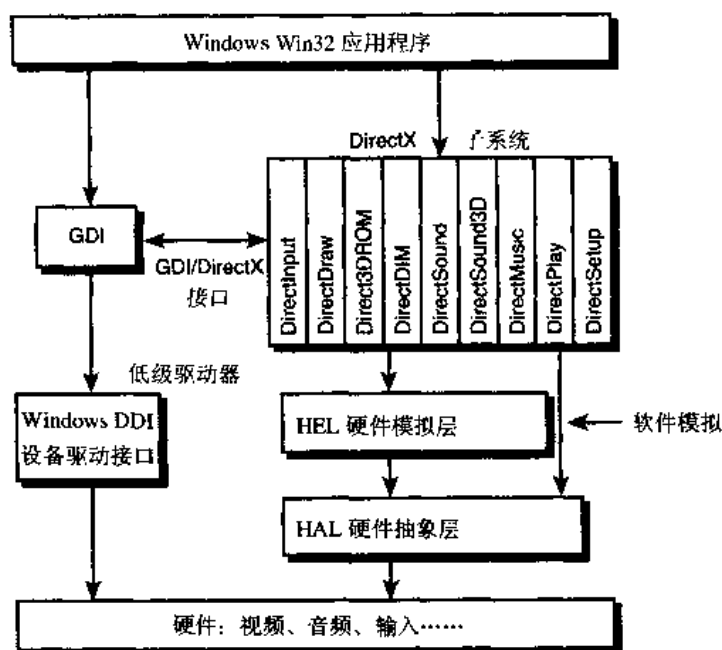


图 5.2 DirectX 结构及其与 Win32 的关系

HAL（硬件抽象层）是接近硬件的底层。它直接和硬件交流。该层通常有设备生产商提供的设备驱动程序，可以通过常规 DirectX 调用直接和硬件进行联系。使用 HAL 要求硬件能够直接支持所要求的性能，这时使用 HAL，就能够提高性能。例如，当绘制一个位图时，使用硬件要比软件更胜任该项工作。

当硬件不支持所要求的性能时，使用 HEL（硬件仿真层）。我们以使用视频卡旋转一个位图为例来说明。如果硬件不支持旋转动作，则使用 HEL，软件算法取代硬件。很明显这样做速度要慢一些，但关键是这样做至少不会中断应用程序，应用程序仍然在运行，仅仅是速度慢一些而已。另外，HAL 和 HEL 之间的切换是透明的。如果要求 DirectX 做某工作，而 HAL 直接处理该项工作，也就是硬件直接处理该项工作。否则，HEL 将调用软件仿真来处理该项工作。

你可能认为软件层次太多。这是一个问题，但实际上 DirectX 非常简单，使用时你要做的可能只是一个或两个额外的函数调用。购买 2D 或 3D 图形、网络 and 音频加速卡花费很

少。你能想像为市场上所有视频加速器来编写一个驱动程序吗？请相信我，这将需要无数的人工作无数年，也就是说这根本就做不到。DirectX 是 Microsoft 和所有的软件开发商倾注了大量努力为你提供的高性能的标准。

深入 DirectX 基础类

下面我们快速浏览一下每个 DirectX6.0 版本的组件，它们是：

DirectDraw——控制视频显示的基本着色和 2D 图形引擎，是所有图形必须穿过的通道，可能是最重要的 DirectX 组件。DirectDraw 的对象大都是系统中的视频卡。

DirectSound——DirectX 中的声音组件，只支持数字声音，不支持 MIDI。但是该组件能够使你的工作简化 100 倍，因为它处理声音时不再需要第三方的声音系统。声音编程是一种黑色艺术，过去没有人想为所有的声卡编写驱动程序。因此，许多生产商目光集中在声音效果库的市场上：Miles Sound System 和 DiamondWare Sound Toolkit。这两个系统只能允许从 DOS 系统或 Win32 平台下装载和播放数字和 MIDI 声音。但是随着 DirectSound、DirectSound3D 以及最新的 DirectMusic 组件的出现，第三方音效库已经使用得越来越少了。

DirectSound3D——DirectSound 的 3D 声音组件，使得 3D 声音在房间中环绕。这是一种相当新的技术，但是正在快速地成熟着。现在大多数声卡支持硬件加速的 3D 效果，包括 Doppler 转换、折射、反射等等。但是如果使用软件仿真的话，这些都不需要了。

DirectMusic——最新的 DirectX 组件，DirectMusic 具有 DirectSound 不支持的 MIDI 技术。不只是这些，DirectMusic 具有全新的可下载声音（DLS）的系统，使得用户能够创建乐器的数字表示方式，并且能够通过 MIDI 控制来回放声音。它就像一个波形表合成器，不过是软件合成器。DirectMusic 具有各种人工智能系统的全新性能引擎。在实时运行中可以通过支持的模板改变音乐。重要的是该系统能够任意创建音乐。

DirectInput——该系统处理所有的输入设备，包括鼠标、键盘、游戏控制杆、操作杆、控制球等等。现在 DirectInput 能够支持力反馈设备，该设备具有电机传动设备和应力传感器，以允许人工显示力，因此用户能够感觉到力的存在。它正在真正地将计算机空间的“性”产业置于过载状态。

DirectPlay——这是 DirectX 的网络方面内容，允许通过 Internet、调制解调器、直接连接或任何其他能够进行交流的媒体进行抽象联系。最酷的是 DirectPlay 使得对网络毫无了解的人也能建立连接，不必编写驱动程序、使用接口等等。另外，DirectPlay 支持会话的概念（正在发展中的游戏）、休息室的概念（游戏玩家聚集和玩的地方）。DirectPlay 不会强制玩家进入任何多人玩的网络结构，它只是为你传递和接收包裹。关于 DirectPlay 的内容和安全性方面已经超出本书范围。

Direct3DRM——Direct3D 保留模式，它是高级的、以对象、帧为基础的 3D 系统，能够用来创建 3D 程序。它利用 3D 加速器，但速度并不是最快的。它对于编写预排程序、模型显示或速度极慢的演示是非常有用的。

Direct3DIM——Direct3D 即时模式，它是低级的 DirectX 的 3D 支持。原来由于和 OpenGL 存在许多冲突，几乎不能使用。老版本的即时模式被称为执行缓冲器，主要是数据和设备的序列，描述绘制的场景非常困难。但是，从 DirectX5.0 开始，即时模式能够通过 DrawPrimitive()函数支持大量的类 OpenGL 界面。这样就可以将三角胶片和风扇等传递到着色引擎中，使用函数调用而不是执行缓冲器来改变状态。因此现在我非常喜欢 Direct3D 即时模式。尽管本卷和第二卷都是关于软件支持的 3D 游戏内容的书，为了完整起见，我们将在第二卷的最后来讨论一下 Direct3DIM。实际上，在第二卷的 CD 上有完整的关于 Direct3D 即时模式的计算机手册。

DirectSetup/Autoplay——这两种组件是准 DirectX 组件，允许一个程序从用户计算机上的应用程序安装 DirectX，当该 CD 放入系统中时立即启动该程序。DirectSetup 是一组很小的在用户计算机上装载的运行时间 DirectX 文件，并在注册表上注册这些文件。AutoPlay 是标准的 CD 子系统，在 CD 根目录上查找 AUTOPLAY.INF 文件。如果找到该文件，AutoPlay 执行该文件中的批处理命令函数。

最后，你可能疑惑 DirectX 为什么有如此多的版本。DirectX 好像每 6 个月升级一次。因为图形和游戏技术变化得太快了，所以我们从事的工作具有一定的冒险性。然而，因为 DirectX 是建立在 COM 技术基础上的，我们为之编写的程序——如在 DirectX3.0 版本上编写的程序在 DirectX7.0 版本上也能运行。下面我们看一下 DirectX 如何工作……

COM：这是 Microsoft 的工作，还是魔鬼的？

当前编写的计算机程序很容易达到几百万行，大一点的系统几乎能达到上亿行程序代码。随着程序不断庞大，编写摘要和体系结构就具有非常重要的意义。否则，将会导致全盘混乱。

C++ 和 Java 是计算机语言方面两种最新的尝试，即使用更多的面向对象的编程技术。C++ 实际上是 C 语言的演变（或者说是回流），在 C 语言中添加了面向对象技术。而 Java 则是建立在 C++ 基础上的，但它是完全面向对象的，并且更为简洁。另外，Java 更大程度上是一个平台，而 C++ 只是一种语言。

虽然使用哪种语言很重要，但归根到底，决定性的是如何用它。尽管 C++ 已经具有了非常酷的面向对象的技术特征，许多人还是不愿使用，或者错误地使用。因此编写大程序依然存在问题。这就是 COM 模型碰上的一个困难。

COM 很多年前作为一种新的软件范例以简单的白皮书的形式出现，这和计算机芯片或 Lego 块工作的方式相类似。只要将它们拼到一起，就可以工作了。计算机芯片和 Lego 块知道如何成为计算机芯片和 Lego 块，因此什么问题都解决了。要利用软件实现这种技术，要使用非常类似的界面，你所能想像的任何类型的函数集都具有它统一的规格。这就是 COM 做的事情。

对于计算机芯片而言，最大的好处就是当你添加更多的芯片时，不必通知其他的芯片。但是如你所知，使用软件程序来完成就困难一点。你至少要重新编译形成可执行文件，解决这一问题是 COM 的另一个目标。你应当能够向 COM 对象添加新的特征，而不会中断使用旧的 COM 对象的软件。另外，COM 对象改变后不必重新编译原来的程序就能工作，这是其最酷的技术特性。

因为能够不进行重新编译程序就可以升级 COM 对象，所以不使用修补程序或新版本就能够升级软件。例如，有这样一个程序，使用三个 COM 对象，分别实现图像、声音和网络化（见图 5.3）。现在假设该软件程序已经售出了 100000 套，但是又不希望发送 100000 套的升级版！要升级图形 COM 对象，那就给用户一个新的图形 COM 对象，程序将自动使用该图形对象。不需要重新编译，不需要连接，一切都不需要，非常简单。当然该技术使用低级语言因而非常复杂，并且需要编写自己的 COM 对象，但是使用起来则非常简单。

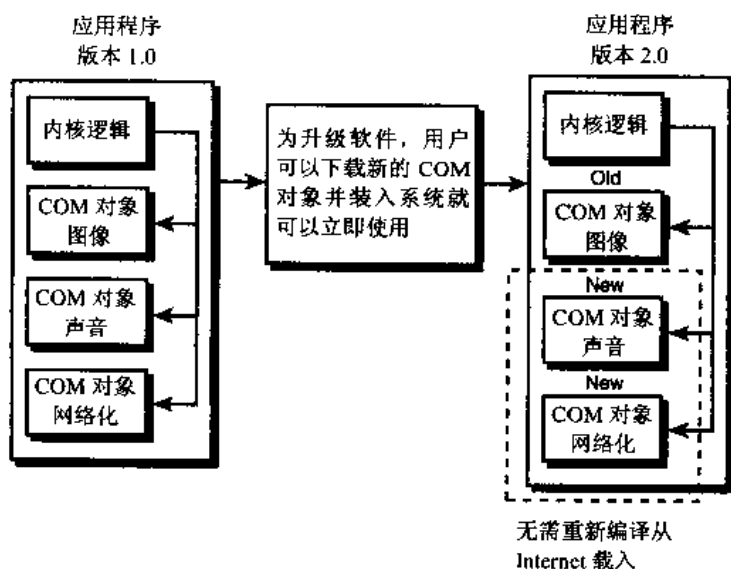


图 5.3 COM 概貌

下一个问题是 COM 对象如何分布或包含，如何体现其即插即用的特性。答案是无章可循，但是在大多数情况下，COM 对象都是 DLL，即动态链接库，随着使用的软件带来或者是下载。这样便于升级和更改。这样惟一的问题是使用该 COM 对象的软件必须知道如何从 DLL 中装载该对象。该部分内容我们将在本章“创建一个准 COM 对象”部分中讨论。

COM 对象究竟是什么？

一个 COM 对象实际上就是实现大量界面的一个 C++ 类或者是一套 C++ 类（一个界面就是一套函数）。这些界面用于和该 COM 对象进行联系，如图 5.4 所示。图中可以看到一个 COM 对象，它分别具有 IGRAPHICS、ISOUND 和 IINPUT 三个界面。

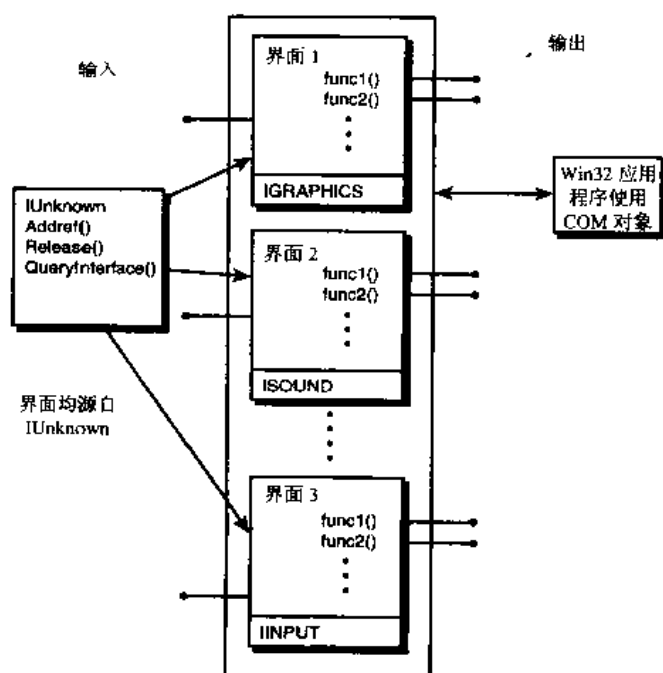


图 5.4 COM 对象的界面

每一个界面都有大量的函数可以调用（只要你知道如何使用）。因此一个 COM 对象可以拥有一个或多个界面，你可以有一个或多个 COM 对象。并且，COM 技术要求说明用户创建的所有的界面必须从一个指定的基本类界面 IUnknown 中导出。对于一个 C 程序员来讲，这就意味着 IUnknown 就像是一个创建界面的字符串指针。

下面是 IUnknown 的类定义：

```
struct IUnknown
{
    // this function is used to retrieve other interface
    virtual HRESULT_stdcall QueryInterface(const IID &iid, (void**)ip) = 0;
    // this is used to increment interface reference count
    virtual ULONG_stdcall AddRef() = 0;
    // this is used to decrement interface reference count
    virtual ULONG_stdcall Release() = 0;
};
```

注意

注意上述所有的方法都是安全而有效的。另外，这些方法使用_stdcall时尊重标准 C/C++ 调用习惯。我们在第二章“Windows 编程模型”中曾经提到过，_stdcall 就是将参数从右向左推进堆栈的。

即使是一个 C++ 程序员，如果对虚函数不熟悉的话，该类定义看上去也有点麻烦。那么我们不选 IUnknown，会发生什么事情呢？IUnknown 导出的所有界面至少要实现

QueryInterface()、AddRef()和 Release()这三个方法。

QueryInterface()对于 COM 非常重要，用来申请一个指向你所希望的界面函数的指针。要实现该请求，必须具有一个界面标识符。该标识符是惟一的指定给界面的数字，长度是 128 位。这样就有 2^{128} 种不同的可能的界面标识符，我敢肯定即使是地球上所有的人什么都不干，都来做一个 COM 对象，花 1 亿年也做不完。我们在本章后面内容中将给出一个实例来详细讨论界面标识符。

COM 的一个规则是：如果已经存在了一个界面的话，可以一直从该界面中申请其他的界面，条件是该界面来自于同一个 COM 对象。也就是说，从一个地方能够到达其他任何地方。图 5.5 给出了其示意图。

AddRef()是一个古怪的函数。COM 对象使用参数计数的技术来跟踪它们的情况。这是由于 COM 的一个声明决定的：该技术和编程语言无关。因此，AddRef()在 COM 对象和界面创建时调用，用来跟踪指向该对象的参数的数目。如果 COM 对象使用 malloc()或 new[]函数的话，那它就和 C/C++ 语言有关。当该引用计数递减到 0 时，该对象就在函数内部消失。

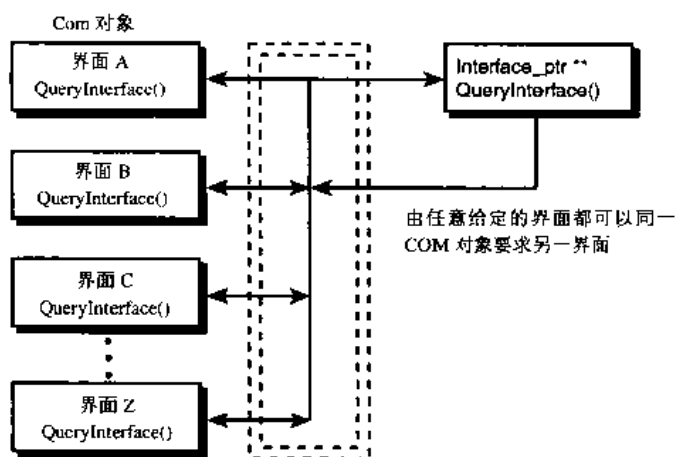


图 5.5 定位一个 COM 对象的界面

技巧

通常，程序员都不必自己在界面上或 COM 对象中调用 AddRef()函数，而由 QueryInterface()函数内部调用 AddRef()函数。但有时候，如果想增加跟踪 COM 对象的参数计数，认为 COM 对象的参数应该更多，而实际上并没有那么多参数时，就必须调用 AddRef()。

这样就带来一个问题，如果 COM 对象是 C++ 类，这些对象在 Visual Basic、Java、ActiveX 等语言中如何创建并使用呢？这只是 COM 的设计者使用虚拟 C++ 类来实现 COM，而不是要求用户也必须使用 C++ 来访问或者是创建它们。只要创建相同的二元图形，当 Microsoft C++ 编译器在创建一个虚拟的 C++ 类时，COM 对象将是和 COM 兼容的。当然，大多数编译器都有各自的工具来协助创建 COM 对象，因此这并不是一个很严重的问题。最酷的是用户能够以 C++、Visual Basic 或 Delphi 来编写一个 COM 对象，而该 COM 对象能够被这

三种中的任何其他语言使用。内存中的二元图形只是内存中的一个二元图形而已。

Release()函数用于减少 COM 对象或界面中的计数参数。大多数情况下，当用户使用一个界面结束时，应当调用该函数。但是有时候，如果用户创建一个对象，并且又从该对象创建了其他的对象，在父对象中调用 **Release()**函数就将会缓慢执行，并首先释放子对象或导出的对象。但是无论是哪种情况，以查询的相反次序使用 **Release()**释放 COM 对象是更好的做法。

界面标识符和 GUID 的详细内容

前面提到，每一个 COM 对象以及界面都必须具有惟一的、用户能够用来申请或访问的 128 位的标识符。这些数字通常称之为 **GUID**（全局单值标识符）。更明确一点讲，当定义 COM 界面时，它们是界面标识符（或称 **IID**）。要生成这些标识符，必须使用 Microsoft 创建的 **GUIDGEN.EXE** 程序（或者使用相同算法编写的类似程序）。图 5.6 表示了运行中的 **GUIDGEN.EXE** 程序。

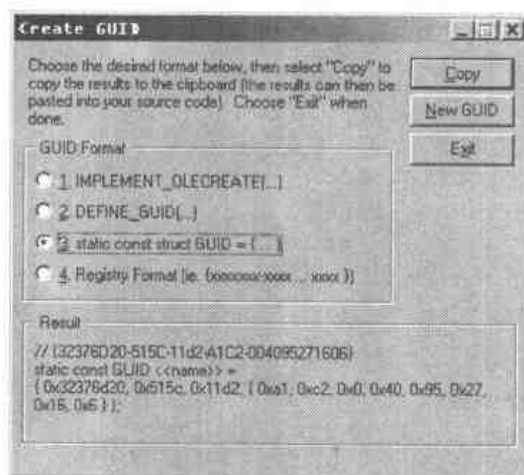


图 5.6 运行中的 GUID 生成器——GUIDGEN.EXE 程序

你要做的就是选择你喜欢的一种标识符格式（有四种不同的格式），然后程序生成一个 128 位的、确保在任何时候任何计算机中都没有创建过的矢量。看上去不可能是吧？不，的确是这样的。这只是一个数学和概率理论的问题。至少该标识符能够工作，因此就不必绞尽脑汁地去刨根问底了。

生成 GUID 或 IID 之后，该标识符就放置在剪贴板中，用户可以使用 **Ctrl+V** 将它粘贴到程序中。下面是我编写的一个 IID 的实例：

```
// {C1BCE961-3E98-11d2-A1C2-004095271606}
static const<<name>>=
{0xc1bce961, 0x3e98, 0x11d2,
{0xa1, 0xc2, 0x0, 0x40, 0x95, 0x27, 0x16, 0x6 } };
```

当然，在你的程序中可以使用你为 GUID 选择的名称来替换<<name>>，但是一定要掌

握这个方法的思路。

GUID 和 IID 都是用来引用 COM 对象及其界面的。因此无论何时创建一个新的 COM 对象及其一套界面，你都必须给使用该 COM 对象的程序员一个惟一的数字。一旦程序员拥有了 IID，就可以创建 COM 对象及其界面。

创建一个准 COM 对象

创建一个完全的 COM 对象超出了本书的内容。用户只需要了解如何使用 COM 对象就行了。当然，如果用户像我的话，应当了解一下如何来创建该对象。因此下面我们就创建一个非常基本的 COM 实例，来帮助用户了解已经创建的 COM 对象的一些问题。

所有的 COM 对象都含有大量的界面，但是所有的 COM 对象都必须首先从 IUnknown 类来导出。然后，一旦建立了所有的界面，就可以将它们放入一个界面容器类中来实现一切。下面我们创建一个具有三个界面 ISound、IGraphics 和 IInput 的 COM 对象来作为一个实例，下面首先是如何定义界面：

```
// the graphics interface
struct IGraphics : IUnknown
{
    virtual int InitGraphics(int mode) =0;
    virtual int SetPixel(int x, int y, int c) =0;
    // more methods...
};

// the sound interface
struct ISound : IUnknown
{
    virtual int InitSound(int driver) =0;
    virtual int PlaySound(int note, int vol) =0;
    // more methods...
};

// the input interface
struct IInput : IUnknown
{
    virtual int InitInput(int device) =0;
    virtual int ReadStick(int stick) =0;
    // more methods...
};
```

现在我们已经有了所有的界面，开始创建界面容器类，该类实际上是 COM 对象的核心：

```
class CT3D_Engine: public IGraphics, ISound, IInput
```

```

{
public:

// implement IUnknown here
virtual HRESULT __stdcall QueryInterface(const IID& iid,
                                         (void **)ip)

{ /* real implementation */ }

// this method increases the interfaces reference count
virtual ULONG __stdcall Addref()
    { /* real implementation */}

// this method decreases the interfaces reference count
virtual ULONG __stdcall Release()
    { /* real implementation */}

// note there still isn't a method to create one of these
// objects...

// implement each interface now

// IGraphics
virtual int InitGraphics(int mode)
    { /*implementation */}
virtual int SetPixel(int x, int y, int c)
    { /*implementation */}

// ISound
virtual int InitSound&(int driver)
    { /*implementation */}
virtual int PlaySound&(int note, int vol)
    { /*implementation */}

// IInput
virtual int InitInput(int device)
    { /*implementation */}

virtual int ReadStick(int stick)
    { /*implementation */}

private:

// .. locals

};

```

注 意



你还是错过了一个创建 COM 对象的类方法。这是个问题。COM 的定义表明创建 COM 对象有许多方法，但没有任何一种方法局限于制定的语言或平台。较简单的一种方法就是通过创建一个 `CoCreateInstance()` 或 `ComCreate()` 函数来创建该对象的一个初始 `IUnknown` 实例。该函数通常装载一个含有 COM 程序代码的、并从该代码开始工作的动态连接库。该技术已经超出了本书内容，我们将在其他地方讨论。

从该实例可以看出，COM 界面和编码实际上和常用的稍微高级一点的 C++ 虚类区别并不大。但是必须合理地创建真正的 COM 对象，要在注册表中注册，并遵循其他的规则。最低限度上说，或多或少它们只是最使用函数指针方法的类（对于 C 程序员来讲，就是结构）。好啦，让我们简要回顾一下 COM 的有关内容。

COM 的简要说明

COM 是编写组件软件的一种新方式——允许创建可重复使用的、能够在运行时间中自动连接的软件模块。每个 COM 对象都具有一个或多个进行实际操作的界面。这些界面无非是通过一个虚函数表单指针来进行访问的方法或函数的集合（在下一章中将详述）。

每一个 COM 对象及其界面和其他对象相比都是惟一的，这是因为在创建 COM 对象及其界面时使用了全程统一标识符 GUID。用户必须使用 GUID 或 IID 来访问 COM 对象及其界面，其他程序员也应如此。

如果创建了一个对旧对象进行升级的新的 COM 对象，应当同时执行旧界面和添加的新界面。有一条非常重要的规则：所有的基于 COM 对象的程序仍然能基于新版 COM 对象运行，而不需要进行重新编译。

COM 是一个通用的定义，能够用于任何计算机以及任何编程语言。惟一的规则就是 COM 对象的二元图形必须是由 Microsoft VC 编译器生成的虚类，因为它只能以该种方式运行。但是 COM 可以应用于其他计算机，如 Mac、SGI 等等，只要遵循这些规则，就可以使用 and 创建 COM 对象。

最后，COM 通过利用基于组件类属的架构开辟了编写大型计算机程序（几亿行程序的规模）的可能性。当然，DirectX、OLE 和 ActiveX 都是基于 COM 技术的，因此应当掌握 COM 技术！

COM 程序实例

DEMO5_1.CPP 是一个完整的创建 COM 对象及其一系列界面的实例。该程序创建了一个由两个界面 IX 和 IY 构成的 COM 对象 CCOM_OBJECT。该程序很好地实现了一个 COM 对象，当然该程序中缺少了一些高级细节，如自动装载的 DLL 等等，但还是完全创建了 COM 对象，并且设计了所有的方法和 `IUnknown` 类。

希望你能够认真阅读该程序，并且能够深入进去，了解该程序如何运行。程序清单 5.1 列出了该 COM 对象和一个简单的 C/C++ 的 `main()` 测试程序的完整的源程序。

程序清单 5.1 完整的 COM 对象程序

```

// DEMO5_1.CPP - A ultra minimal working COM example
// NOTE: not fully COM compliant

// INCLUDES //////////////////////////////////////

#include <stdio.h>
#include <malloc.h>
#include <iostream.h>
#include <objbase.h> // note: you must include this header it
                    // contains important constants
                    // you must use in COM programs

// GUIDS //////////////////////////////////////

// these were all generated with GUIDGEN.EXE

// {B9B8ACE1-CE14-11d0-AE58-444553540000}
const IID IID_IX =
{ 0xb9b8ace1, 0xce14, 0x11d0,
  { 0xae, 0x58, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };

// {B9B8ACE2-CE14-11d0-AE58-444553540000}
const IID IID_IY =
{ 0xb9b8ace2, 0xce14, 0x11d0,
  { 0xae, 0x58, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };

// {B9B8ACE3-CE14-11d0-AE58-444553540000}
const IID IID_IZ =
{ 0xb9b8ace3, 0xce14, 0x11d0,
  { 0xae, 0x58, 0x44, 0x45, 0x53, 0x54, 0x0, 0x0 } };

// INTERFACES //////////////////////////////////////

// define the IX interface
interface IX: IUnknown
{

virtual void __stdcall fx(void)=0;

};

// define the IY interface
interface IY: IUnknown
{

virtual void __stdcall fy(void)=0;

};

```

```
// CLASSES AND COMPONENTS //////////////////////////////////////

// define the COM object
class CCOM_OBJECT : public IX,
                    public IY
{
public:

    CCOM_OBJECT() : ref_count(0) {}
    ~CCOM_OBJECT() {}

private:

    virtual HRESULT __stdcall QueryInterface(const IID &iid, void **iface);
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    virtual void __stdcall fx(void)
        {cout << "Function fx has been called." << endl; }
    virtual void __stdcall fy(void)
        {cout << "Function fy has been called." << endl; }

    int ref_count;

};

// CLASS METHODS //////////////////////////////////////

HRESULT __stdcall CCOM_OBJECT::QueryInterface(const IID &iid,
                                              void **iface)
{
    // this function basically casts the this pointer or the IUnknown
    // pointer into the interface requested, notice the comparison with
    // the GUIDs generated and defined in the beginning of the program

    // requesting the IUnknown base interface
    if (iid==IID_IUnknown)
    {
        cout << "Requesting IUnknown interface" << endl;
        *iface = (IX*)this;

        } // end if

    // maybe IX?
    if (iid==IID_IX)
    {
        cout << "Requesting IX interface" << endl;
        *iface = (IX*)this;

        } // end if
}
```



```

else // maybe IY
if (iid- IID_IY)
{
    cout << "Requesting IY interface" << endl;
    *iface = (IY*)this;

    } // end if
else
{ // cant find it!
    cout << "Requesting unknown interaface!" << endl;
    *iface = NULL;
    return(E_NOINTERFACE);
    } // end else

// if everything went well cast pointer to
// IUnknown and call addref()
((IUnknown *) (*iface))->AddRef();

return(S_OK);

} // end QueryInterface

////////////////////////////////////

ULONG __stdcall CCOM_OBJECT::AddRef()
{
    // increments reference count
    cout << "Adding a reference" << endl;
    return(++ref_count);

} // end AddRef

////////////////////////////////////

ULONG __stdcall CCOM_OBJECT::Release()
{
    // decrements reference count
    cout << "Deleting a reference" << endl;
    if (--ref_count==0)
    {
        delete this;
        return(0);
    } // end if
else
    return(ref_count);

} // end Release

////////////////////////////////////

IUnknown *CoCreateInstance(void)
{

```

```
// this is a very basic implementation of CoCreateInstance()
// it creates an instance of the COM object, in this case
// I decided to start with a pointer to IX -- IY would have
// done just as well

IUnknown *comm_obj = (IX *)new(CCOM_OBJECT);

cout << "Creating Comm object" << endl;

// update reference count
comm_obj->AddRef();

return(comm_obj);

} // end CoCreateInstance

////////////////////////////////////

void main(void)
{

// create the main COM object
IUnknown *punknown = CoCreateInstance();

// create two NULL pointers the IX and IY interfaces
IX *pix=NULL;
IY *piy=NULL;

// from the original COM object query for interface IX
punknown->QueryInterface(IID_IX, (void **)&pix);

// try some of the methods of IX
pix->fx();

// release the interface
pix->Release();

// now query for the IY interface
punknown->QueryInterface(IID_IY, (void **)&piy);

// try some of the methods
piy->fy();

// release the interface
piy->Release();

// release the COM object itself
punknown->Release();

} // end main
```

我已经对该程序进行了预编译，形成了可执行文件 DEMO5_1.EXE。当然，如果读者想测试和编译 DEMO5_1.CPP 的话，请记住创建一个 Win32 控制台应用程序，因为该演示程序使用 main() 函数而不是 WinMain() 函数，并且这只是一个基于文本的程序。

应用 DirectX COM 对象

现在已经对 DirectX 以及 COM 的工作原理有了一个初步的认识，下面看一下 DirectX 和 COM 怎样协同工作。如前面所述，构成 DirectX 的 COM 对象有很多。这些 COM 对象在调用 DirectX 的运行时间版本时作为动态连接库包含在系统中。当运行一个第三方的 DirectX 游戏时，DirectX 应用程序装载一个或多个动态连接库，然后出现界面，并且这些界面的程序（函数）开始工作。这是运行时间方面的情况。

编译时间方面略有不同。DirectX 设计人员知道主要是游戏程序员使用 DirectX，并且认为大部分游戏程序员都不喜欢使用 Windows 编程——这倒是真的。他们知道应当尽量将 COM 内容减小到最小的程度，否则游戏程序员也不喜欢使用 DirectX。因此，90% 的 DirectX COM 对象都封装了极少的只和 COM 装载有关的函数调用。因此不需要调用 CoCreateInstance() 函数，只要像上面那样进行 COM 初始化和装载即可。但还应当使用 QueryInterface() 函数搜索到新界面，我们对此应当有所了解。关键问题是 DirectX 尽量使用户从 COM 繁琐使用中解脱出来，因此用户可以使用 DirectX 的核心功能进行工作。

如上所述，要编译一个 DirectX 程序，应当包含大量的封装了 COM 对象的输入库函数，以便于能够使用这些封装函数调用 DirectX 创建 COM 对象。大多数情况下，我们需要下面的库函数：

```
DDRAW.LIB
DSOUND.LIB
DINPUT.LIB
DSETUP.LIB
DPLAYX.LIB
D3DIM.LIB
D3DRM.LIB
```

请记住，这些库函数本身并不包含 COM 对象。它们仅仅是封装库函数，用于调用、装载 DirectX 动态连接库，DLL 才是 COM 对象。最后，当调用一个 DirectX COM 对象时，仅仅是调用一个界面指针而已，就是开始操作的位置。就像在 DEMO5_1.CPP 程序实例中，一旦有了界面指针，就可以任意地进行函数调用，或者用 C++ 语言更准确地说就是方法调用。C 程序员如果感觉对函数指针不习惯的话，可以迅速浏览下一部分的内容。而对于一个 C++ 程序员来讲，则可以直接跳过下一部分的内容。

COM 和函数指针

一旦创建了一个 COM 对象，并且获取了一个界面的指针，实际上就是一个 VTABLE（虚函数表）指针。图 5.7 给出了其示意图。使用虚函数主要是为了在到达运行时间之前无约束地使用函数调用。这对于 COM 和虚函数来讲是至关重要的。总的来讲，C++ 内嵌了该虚函数，同样程序员也能够使用 C 语言通过应用显式函数指针来进行函数调用。

一个函数指针就是一个用来调用函数的指针类型。尽管该函数和一些代码密切相关，但是只要该函数原型指针和指向的函数相同，就可以对该函数任意操作。例如说，编写一个图像驱动函数在屏幕上绘制一个点。但是可能有许多不同的显卡来支持该功能，这些显卡的工作原理各不相同，如图 5.8 所示。

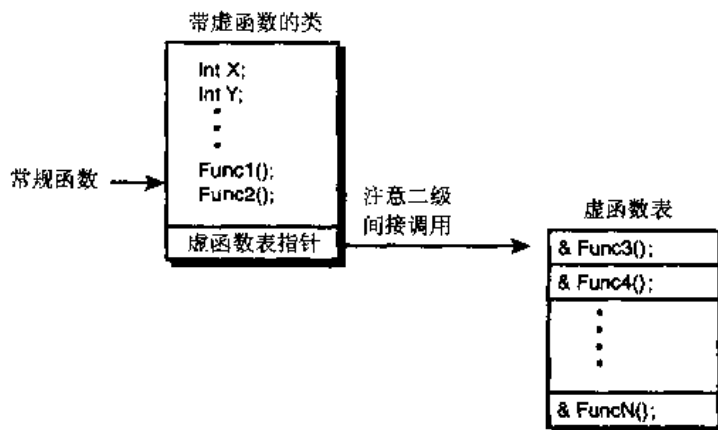


图 5.7 虚函数表架构

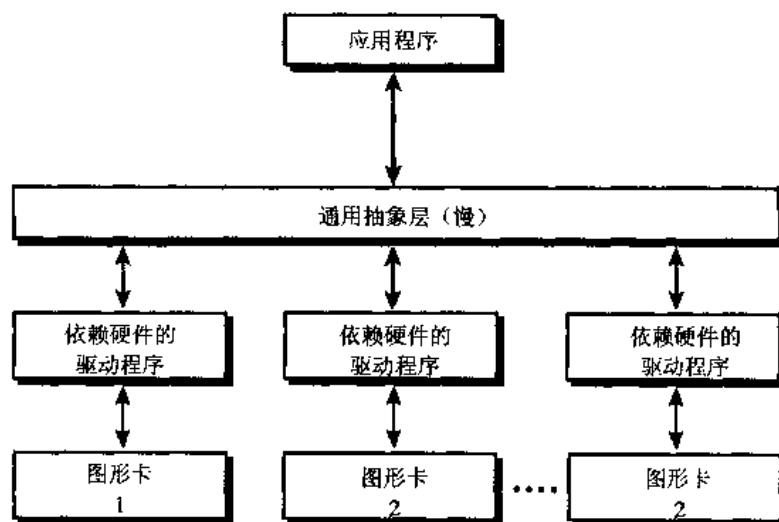


图 5.8 支持不同显卡的软件设计

我们希望对于所有的显卡使用相同的方式调用绘制像素函数，但内部代码由于插入了不同的显卡而不同。下面是典型的 C 程序员的解决方法：

```
int SetPixel(int x, int y, int color, int card)
{
    // what video card do we have?
    switch(card)
    {
        case ATI:      /*hardware specific code */ } break;
        case VOODOO:   /*hardware specific code */ } break;
        case SIII:     /*hardware specific code */ } break;
        .
        .
        .
        default:       /*standard VGA code */ } break;

    } // end switch

    // return success
    return(1)

} // end SetPixel
```

看出上面程序中的问题了吗？首先，switch 语句效率不高，速度缓慢、语句冗长并且容易出错，在添加支持其他显卡时还要中断函数。使用标准 C 语言的更好的解决方法是使用函数指针，如下所示：

```
// function pointer declaration, weird huh?
int (*SetPixel)(int x, int y, int color);
// now here's all our set pixel functions

int SetPixel_ATI(int x, int y, int color)
{
    // code for ATI
} // end SetPixel_ATI

////////////////////////////////////

int SetPixel_VOODOO(int x, int y, int color)
{
    // code for VOODOO

} // end SetPixel_VOODOO

////////////////////////////////////

int SetPixel_SIII(int x, int y, int color)
{
    // code for SIII

} // end SetPixel_SIII
```

这样一切都 OK 了！系统启动时，程序检查安装了哪一种显卡，然后就设置特定的函数指针指向准确的显卡函数。例如，如果想令 SetPixel()函数指向 ATI 显卡函数，可以像下面这样编写代码：

```

// assigning a function pointer
SetPixel = SetPixel ATI;

```

非常简单！图 5.9 给出了其示意图。

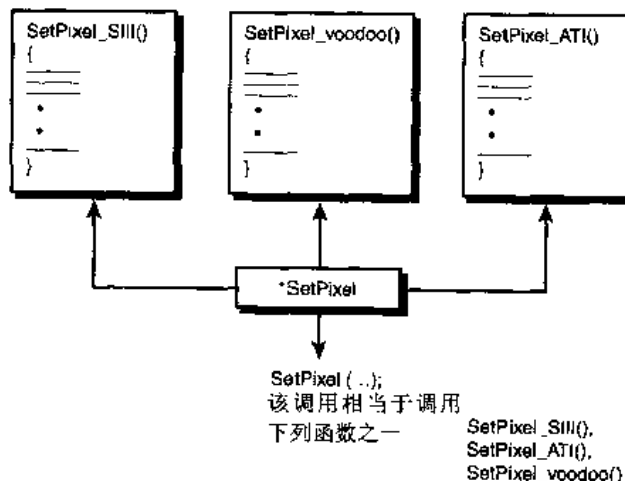


图 5.9 使用函数指针编写不同的代码块

注意，SetPixel()函数在某种程度上讲是 SetPixel_ATI()函数的别名。这对于函数指针来讲是至关重要的。要调用 SetPixel()函数，就要进行标准调用，而不是调用一个空函数 SetPixel()，该调用实际上就是调用 SetPixel_ATI()函数：

```
// this really calls SetPixel_ATI(10,20,4)
SetPixel(10,20,4)
```

关键问题是代码要前后一致，但是对于如何指派函数指针则是另外一件事了。该技术就是这样酷，许多 C++ 和虚函数都是建立在该技术基础上的。也就是说所有的虚函数实际上都是函数指针的后结合法，并且完美地嵌入了编程语言中，然后就组合到一起，和上面所做的一样。

下面我们讨论一下如何完成通用显卡驱动程序的连接，然后测试已安装的显卡，设定 SetPixel()函数指针指向正确的 SetPixel*()函数。如下所示：

```
int SetCard(int card)
{
    // assign the function pointer based on the card
    switch (card)
    {
```

```

        case ATI:
        {
            SetPixel = SetPixel_ATI;

            } break;

        case Voodoo:
        {
            SetPixel = SetPixel_Voodoo;
            } break;

        case SIII:
        {
            SetPixel = SetPixel_SIII;
            } break;

        default: break;

    } // end switch

} // end SetCard

```

在程序开始部分，应当建立一个安装函数的调用，如下所示：

```
SetCard(card);
```

剩下的工作就很好做了。这就是函数指针和虚函数在 C++ 中的使用方法，因此下面我们看一下该技术如何在 DirectX 中使用。

创建和使用 DirectX 界面

关于这一点，我认为读者应当理解为 COM 对象是界面类集，该界面集只是函数指针（更准确地说是 VTABLE）。因此使用 DirectX COM 对象来编程只要创建 COM 对象，获得一个界面指针，然后使用正确的语法调用该界面即可。下面使用 DirectDraw 主界面为例来示范使用 DirectX COM 对象编程的过程：

首先，需要对 DirectDraw 进行下面三方面工作：

- DirectDraw 运行时间 COM 对象和动态连接库必须进行装载并注册。这由 DirectX 安装程序来完成。
- 必须包含 Win32 程序中的 DDRAW.LIB 输入库，以便于你调用的封装函数的连接。
- 必须包含 Win32 程序中的 DDRAW.H 文件，以便于编译器明白 DirectDraw 的头文件信息、原型以及数据类型。

下面是 DirectDraw 界面指针的数据类型：

```
LPDIRECTDRAW lpdd =NULL;;
```

要创建 DirectDraw COM 对象及检索指向 DirectDraw 对象的界面指针（该对象表示显

卡), 应当使用封装函数 `DirectDrawCreate()`, 如下所示:

```
DirectDrawCreate(NULL, &lpdd, NULL);
```

在第六章“首次接触: DirectDraw”中, 将详细讨论其参数。现在, 只需要了解该调用创建了一个 `DirectDraw` 对象, 并且指定界面指针指向 `lpdd` 就可以了。

注 意



当然, 在该函数中还有许多内容需要进行。该函数打开一个动态连接库, 装载该 DLL, 建立调用以及进行其他的众多动作。但是你不必为此操心。

下面的工作就是具体的操作, 并且能够调用 `DirectDraw`。但现在还不知道可用的方法或函数, 这就是你继续阅读本书的原因了。下面是如何将显示模式设定为 640×480 、256 色的实例:

```
lpdd->SetVideoMode(640, 480, 256);
```

非常简单, 是吧? 剩下的惟一的任务就是废除 `DirectDraw` 界面指针 `lpdd`。当然只要查看一下界面虚表即可, 不必为此担心。

本质上讲, 任何 `DirectX` 的调用都采用下面方式:

```
interface_pointer->method_name(parameter list);
```

当然也可以从原始 `DirectDraw` 界面通过应用 `QueryInterface()` 函数来使用其他的任何想使用的界面, 如 `Direct3D`。由于目前 `DirectX` 有许多版本, 不久之前 Microsoft 刚刚停止编写检索最新版本界面的封装函数。这就意味着必须使用 `QueryInterface()` 函数人工检索最新的 `DirectX` 界面。我们简单了解一下该方面内容。

界面查询

关于 `DirectX` 有件古怪的事情, 就是所有的 `DirectX` 版本都是不同步的。这样就存在一个问题, 容易造成混乱。下面是解决该问题的方法: 当使用 `DirectX1.0` 版本时, `DirectDraw` 界面应当像下面方式这样命名:

```
IDIRECTDRAW
```

当使用 `DirectX2.0` 版本时, `DirectDraw` 升级为 2.0 版本, 应当如下命名:

```
IDIRECTDRAW
```

```
IDIRECTDRAW2
```

现在使用 `DirectX6.0` 版本, 应当如下命名:

```
IDIRECTDRAW
```


IDIRECTDRAW2

IDIRECTDRAW4

还有，对于界面 3 和 5 怎样命名呢？我也不知道，这就是个问题。因此尽管正在使用 DirectX7.0 版本，也并不意味着界面能够升级到 7.0 版本。并且它们都是不同步的。尽管 DirectX6.0 可以使 DirectDraw 界面升级到 IDIRECTDRAW4，但是 DirectSound 只升级到 1.0 版本的界面，称之为 IDIRECTSOUND。真是糟糕透顶！该问题的主旨就是无论什么时候使用 DirectX 界面，都应当确认正在使用其最新版本。如果不能确认的话，应当使用同类创建函数中的修订版 1 的界面指针来获得其最新版本。

下面是我们正在讨论的问题的一个实例：DirectDrawCreate()函数返回一个修订版 1 的界面指针，而 DirectDraw 已经升级到 IDIRECTDRAW4。应当如何利用这一新功能呢？

提示

如果你对这部分内容不甚清楚，也不必着急。我会用到 Direct 1.0 到 5.0 版的界面，因为 DirectX 这部分的文件实在比较乱——通常都是这样。

有两种方式可以做到这一点：使用低级 COM 函数，或者使用 QueryInterface()函数。下面我们使用后一种方式。过程如下：首先，使用 DirectDrawCreate()函数调用创建 DirectDraw COM 界面。该函数返回一个讨厌的 IDIRECTDRAW 界面指针。然后，使用该指针建立一个 QueryInterface()函数的调用，使用 IDIRECTDRAW4 的界面标识符（GUID）来检索该调用。下面是一个实例：

```
LPDIRECTDRAW lpdd; // version 1.0
LPDIRECTDRAW4 lpdd4; // version 6.0, but called 4.0

// create version 1.0 DirectDraw object interface
DirectDrawCreate(NULL, &lpdd, NULL);

// now look in DDRAW.H header, find IDIRECTDRAW4 interface
// ID and use it to query for the interface
lpdd->QueryInterface(IID_IDirectDraw4, &lpdd4);
```

此时就有了两个界面指针。实际上 IDIRECTDRAW 指针是不需要的，因此要释放它：

```
// release, decrement reference count
lpdd->Release();

// set to NULL to be safe
lpdd = NULL;
```

记住了吗？你应当在完成任务时释放一个界面。因此，当程序终止时，也要废弃

IDirectDraw4 界面，如下所示：

```
// release, decrement reference count
lpdd4->Release();

// set to NULL to be safe
lpdd4 = NULL;
```

这就是使用 DirectX 和 COM 进行编程的全部内容。当然，这里没有列出 DirectX 组件的全部的几百个函数和所有的界面，但是你以后会看到的。

COM 的前景

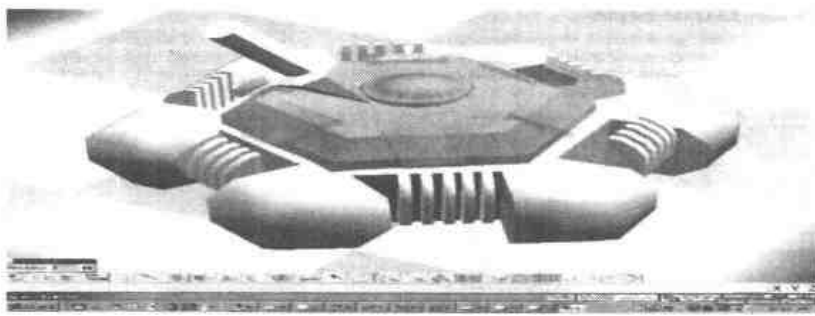
当前已经有许多和 COM 相似的分布对象技术，例如 COBRA（通用对象代理架构）。但是我们关心的是 Windows 游戏，所以其他技术就都无关紧要了。

最新版本的 COM 称为 COM++，其功能更加强大，具有更好的规则和更完善的执行细节。COM++ 采用了更容易创建的分布组件软件。它比 COM 复杂得多，当然，这是 COM 设计者应当作的。

除了 COM 和 COM++ 之外，COM 还有一个完全 Internet/Intranet 版本，称之为 DCOM（分布式 COM）。使用 DCOM 技术，COM 对象就不必再安装在用户的计算机上，可以从网络中的其他计算机中使用。这个技术非常酷吧？假设拥有大型的 DCOM 服务器，程序就可以作为用户来使用。这真是不可思议的技术。

总 结

本章主要讨论了一些纯粹的技术内容和概念。COM 并不容易理解，要真正掌握它必须要花一点时间来学习。但是使用 COM 要比理解它容易 10 倍，下一章中我们将讨论如何使用 COM。另外，我们也浏览了 DirectX 及其所有的组件。因此一旦在下一章中了解了每一个组件的细节以及如何使用，对 DirectX 和 COM 的协同工作原理就会有更深入的了解。



6

首次接触：DirectDraw

在本章中，将第一次接触到DirectX的最重要的组成部分之一：DirectDraw。DirectDraw可能是DirectX中最重要技术，因为它沟通了2D图形的显示和Direct3D所依赖的帧缓冲层。只要掌握了DirectDraw，就能够编写各种在DOS16/32下编写的图形应用程序。DirectDraw是理解DirectX 中许多概念的关键，所以要特别注意。

以下是本章的主要内容：

- ➔ DirectDraw 的界面
- ➔ 创建一个DirectDraw对象
- ➔ 与Windows 协同工作
- ➔ 进入事件模式
- ➔ 巧妙的色彩
- ➔ 建立一个显示画面

DirectDraw 界面

DirectDraw由一些界面组成。如果掌握了第五章关于COM的讨论，即“DirectX 基础和令人生畏的COM”，可以知道界面只不过是一些用来和组件联系的函数和（或）方法的集合体。图6.1给出了DirectDraw 界面的示意图。

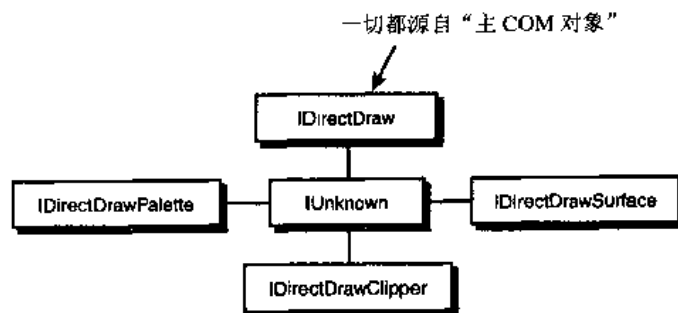


图 6.1 DirectDraw 界面

界面特征

从图中可以看到，DirectDraw 由 5 个界面组成：

IUnknown——所有的 COM 对象必须从这个基本界面中获得，DirectDraw 也不例外。IUnknown 只包含了被其他界面覆盖的 `Addref()`、`Release()` 及 `QueryInterface` 函数。

IDirectDraw——这是一个必须创建出同 DirectDraw 一起开始工作的主要界面对象。IDirectDraw 字面上表示视频卡和支持硬件。有意思的是，现在在 MMS（多监视支持）和 Windows 98/NT 下，你可以在系统中安装多个视频卡，因此就会有多个 DirectDraw 对象。不过，本书中，即使系统中有多个卡，我们也假定在计算机中只有一个视频卡，而且总是选择一个默认卡代表 DirectDraw 对象。

IDirectDrawSurface——这代表你将创建的实际显示画面，显示时需要利用到 DirectDraw。视频卡仅利用本身的 VRAM（视频 RAM）而不用系统的存储器就能存储一个 DirectDraw 画面。

主画面通常是代表正在被视频卡光栅化并显示的实际视频缓冲，另一方面，辅助画面通常不在显示区内。大多数情况下，你只需创建一个最初的画面表示实际视频显示，然后创建多个辅助画面表示对象位图和（或）用反向缓冲代表后备作图区。在此后备作图区中生成下一幅动画帧。在本章的后面部分将对画面的一些细节问题进行介绍。图 6.2 显示了一个小图画的制作。

IDirectDrawPalette——DirectDraw 可以处理任何色空间，从 1 位单色到 32 位真彩色。所以，DirectDraw 支持 IDirectDrawPalette 界面处理使用 256 色或较少彩色的显示方式的调色板。这种情况下，你在一些演示中应多使用 256 色模式，因为对于软件光栅来说，256 色是最快的模式。在第二册中有关 Direct3D 即时模式的讨论中你将转面用 24 位色彩，因为 24 位色彩是 Direct3D 工作的基本方式。本例中，IDirectDrawPalette 界面用来创建、加载、操作调色板及调用调色板来画图，如主画面及辅助画面，图 6.3 表示了画面和 DirectDraw 调色板之间的联系。

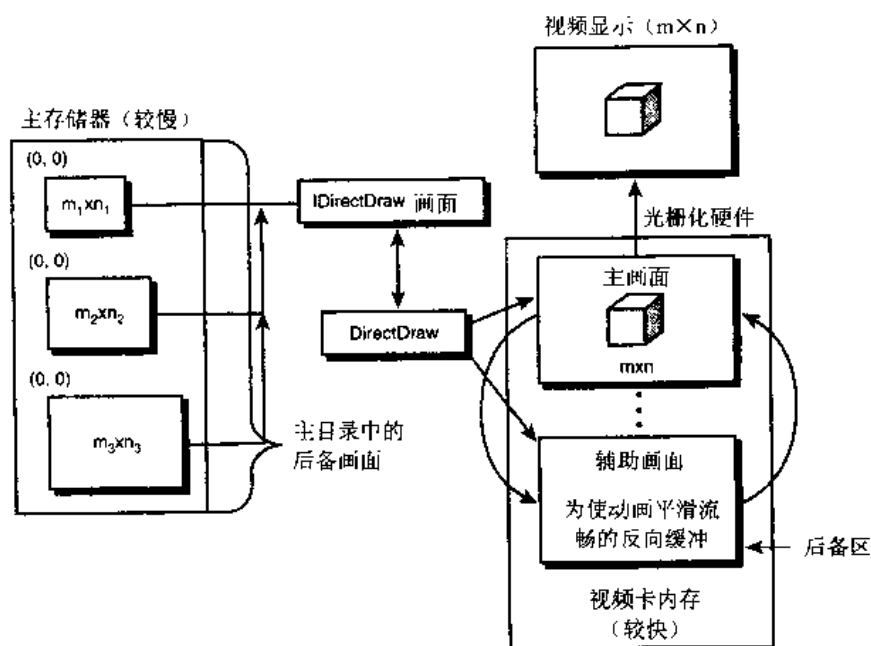


图 6.2 DirectDraw 画面

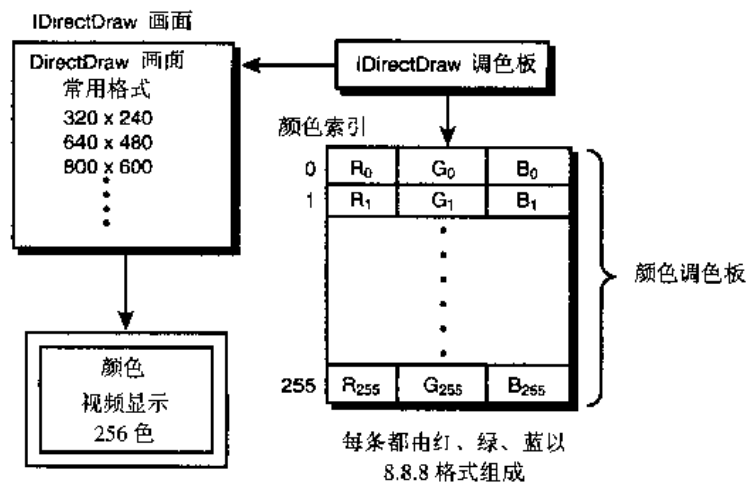


图 6.3 DirectDraw 画面和调色板的关系

IDirectDrawClipper——IDirectDrawClipper用于一些可视显示面的子集剪切DirectDraw光栅和位图操作。大多数情况下，只在视窗DirectX应用程序软件中使用DirectDraw剪切器，和（或）用DirectDraw 剪切器在显示画面的区域里对位图进行剪切。IDirectDrawClipper界面的优点是：如果有相应的硬件，IDirectDrawClipper界面能利用硬件加速，而且可以对在屏幕区域内通常需要剪切位图的像素及图像进行处理。

在创建一个DirectDraw对象前，需要重新回顾一下上一章关于COM的内容。DirectDraw和所有的DirectX组件都处于稳步发展中，界面一直在升级。因此，即使到现在为止，本章

中涉及到 DirectDraw 的界面从类属上说是指 IDirectDraw、IDirectDrawSurface、IDirectDrawPalette 及 IDirectDrawClipper，大部分界面已经更新，而且有了更新的版本。如 IDirectDraw 升级到 DirectX 第6版的 IDirectDraw4。

所以，如果想要得到最近的软件和硬件性能，就需要使用 IUnknown::QueryInterface() 函数获得最新的修订版本。如果想弄清楚，须参看 DirectX SDK 手册。当然，本书中，使用 DirectX 6.0，所以知道哪些是最新的界面，但是，要记住，当升级到7.0版本的时候就会有一些你想要的更新的界面。然而，本书的两部分内容都是关于光栅转换和3D软件的。所以，它是很实用的。在大多数情况下，你在新版本中也几乎用不上别的技巧。

界面的协同使用

下面，将简单介绍用这些界面如何创建一个 DirectDraw 应用程序：

1. 创建主 DirectDraw 对象，得到一个 IDirectDraw4 界面。在此界面上，设置协同等级和视频模式。
2. 在 IDirectDrawSurface 界面上，创建至少一个主画面。基于画面的色深及视频模式，如果视频模式为每像素8位或更少，则需要使用调色板。
3. 在 IDirectDrawPalette 界面上创建一个调色板，用 RGB 三元组初始化调色板，并把调色板附在界面上。
4. 如果 DirectDraw 程序带有一个窗体，或对一个可能出了 DirectDraw 可视界面边界的位图进行着色时，你就需要创建一个剪切工具，并把它调整到可视窗体的区域中。如图6.4所示。

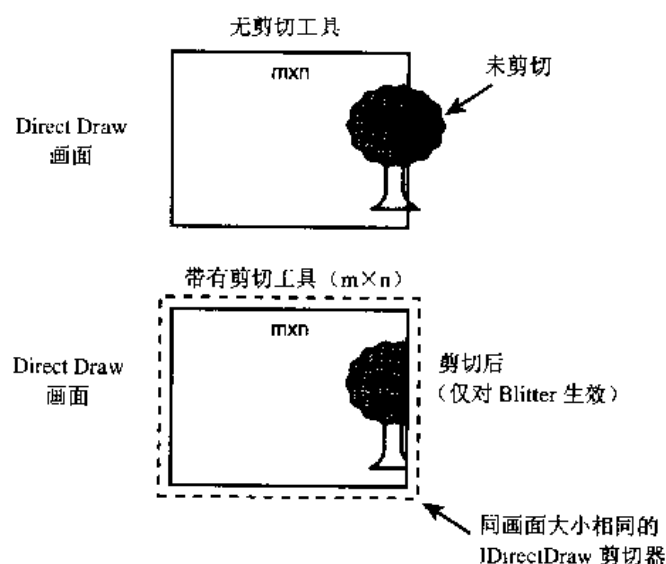


图 6.4 DirectDraw 剪切

5. 在主画面上画图。

当然，上述步骤省去了许多细节，但这是使用不同界面的精髓。下面我们开始认真地探讨细节问题并使这些界面真正地工作。

创建 DirectDraw 对象

用C++创建一个DirectDraw对象，你需要调用DirectDrawCreate()函数，如下：

```
HRESULT WINAPI DirectDrawCreate(GUID FAR *lpGUID, //guid of object
                                LPDIRECTDRAW FAR *lplpDD, //receives interface
                                [unknown FAR *pUnkOuter]); //com stuff
```

lpGUID——这是你所需要的显示驱动的 GUID（全局统一标识符）。大多数情况下，你只须用 NULL 代替默认硬件。

lplpDD——这是一个指向获得IDirectDraw指针的指针，注意：lplpDD返回一个IDirectDraw界面，而不是IDirectDraw4界面。

pUnkOuter——高级特征：通常设置为NULL。

以下是在IDirectDraw界面上创建一个默认DirectDraw对象使用的程序：

```
LPDIRECTDRAW lpdd=NULL; //storage for IDirectDraw

//create the DirectDraw object
DirectDrawCreate(NULL,&lpdd,NULL);
```

如果程序执行成功的话，lpdd将成为一个有效的IDirectDraw对象界面。如果你想得到最新的界面IDirectDraw4，得先了解一下DirectDraw的错误处理。

DirectDraw 的错误处理

DirectX中的错误处理是十分清楚的。有一些能够测试出任何程序或函数执行是否成功的宏。对DirectX函数中的错误进行测试采用微软授权的两个宏。

FAILED()——对错误进行测试。

SUCCEEDED()——对成功进行测试。

在此基础上，应加上以下的错误处理代码：

```
if (FAILED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
    // error
} // end if
```

同样，判断测试是否成功，你可以加上以下代码：

```

if(SUCCEEDED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
    // move on to next step
} // end if
else
{
    // error
} // end else

```

我通常使用 `FAILED()` 宏，因为我不喜欢有两个不同的逻辑路径。但不管怎样。这个宏的主要问题是它本身并不能告诉你很多。它们更倾向于检测常规问题。如果你想知道确切的问题，你可以浏览一下程序返回的代码。表 6.1 列出了 DirectX 6.0 版本中 `DirectDrawCreate()` 函数返回的代码。

表 6.1 `DirectDrawCreate()` 返回的代码

返回代码	说 明
<code>DD_OK</code>	成功
<code>DDERR_DIRECTDRAWALREADYCREATED</code>	DirectDraw 对象已经被创建
<code>DDERR_GENERIC</code>	DirectDraw 不知道哪出错了
<code>DDERR_INVALIDDIRECTDRAWGUID</code>	GUID 设备未知
<code>DDERR_INVALIDDIRECTDRAWGUID</code>	传递的参数有错
<code>DDERR_NODIRECTDRAWHW</code>	没有任何硬件
<code>DDERR_OUTOFMEMORY</code>	大胆地猜猜看

常量和条件逻辑一起使用的惟一问题是，微软并不保证他们不会完全改变所有的错误代码。不过，在任何情况下你可以相当放心地使用以下代码：

```

if (DirectDrawCreate(...) != DD_OK)
{
    // error
} // end if

```

而且，所有 `DirectDraw` 函数定义了 `DD_OK`，所以你可以放心地使用。

改进界面

如上所示，你可以用存储在 `lpdd` 中的基本 `IDirectDraw` 界面调用 `DirectDrawCreate()`。或通过 `IUnknown` 界面的方法 `QueryInterface()` 设法找到新的界面，把界面升级到最新版本（不管它是怎样的）。`QueryInterface()` 是每个 `DirectDraw` 界面的工具。DirectX 6.0 版本的最新 `DirectDraw` 界面是 `IDirectDraw4`，所以，以下是如何获得一个界面指针。


```

LPDIRECTDRAW lpdd=NULL; //standard DirectDraw 1.0
LPDIRECTDRAW lpdd4=NULL; // DirectDraw 6.0 interface 4

// first create base IDirectDraw interface
if (FAILED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
    // error
} // end if

// now query for IDirectDraw4
if (FAILED(lpdd->QueryInterface(IID_IDirectDraw4,
                               (LPVOID *)&lpdd4)))
{
    // error
} // end if

```

有两点需要注意:

- `QueryInterface()`调用的方式。
- 用来申请创建`IDirectDraw4`界面的常量是`IID_IDirectDraw4`。

通常,所有的界面调用都具有以下形式:

```
interface_pointer->method(parms...);
```

所有的界面标识符用以下形式:

```
IID_IDirectCD
```

在此, C 指代组件: Draw 代表 DirectDraw, Sound 指代 DirectSound, Input 代表 DirectInput, 等等。D 是个编号, 从 2 到 n, 表明了你想的界面。另外, 你可以在 `DDRAW.H` 文件中找到这些常量。

继续这个例子, 你可能有点尴尬, 因为你现在有一个 `IDirectDraw` 界面, 一个 `IDirectDraw4` 界面。该怎样做呢? 既然你不需要旧的界面, 就把旧的界面释放掉, 做法如下:

```

lpdd->Release();
lpdd=NULL; // set to NULL for safety

```

从现在开始, 所有的方法调用都使用新的 `IDirectDraw4` 界面。

警告



`IDirectDraw4`的新功能使一切变得有井井有条。这不仅仅是因为`IDirectDraw4`界面更为先进和完善, 而且在许多情况下`IDirectDraw4`界面需要并返回新的数据结构, 而不是DirectX 1.0定义的基本结构。了解这些变化的惟一途径是查看DirectX SDK文献, 检查具体的函数所需的或(和)返回的数据结构的版本。然而, 这只是一般的警告。本书中, 我将给出你要完成的所有例子的正确的结构。

除了在最初的 `IDirectDraw` 界面指针 (`lpdd`) 中使用 `QueryInterface()`函数外, 还有一种可以直接得到 `IDirectDraw` 界面的“COM 方式”。这种方式更为直接。在 COM 下, 只要有界面标识符或称 IID (代表你想要的界面), 你就可以得到任何界面的界面指针。大多数

情况下，我个人不喜欢使用低水平的 COM 函数。

尽管如此，当你接触到 DirectMusic 时，除了使用低水平的 COM 函数外没有其他方法。所以，至少在此适当介绍一下这个过程。通过以下代码，你就可以直接创建一个 IDirectDraw 界面：

```
//first initialize COM, this will load the COM libraries
// if they aren't already loaded
if(FAILED(CoInitialize(NULL)))
{
    // error
} // end if

// Create the DirectDraw object by using the
// CoCreateInstance() function
if (FAILED(CoCreateInstance(&CLSID_DirectDraw,
                           NULL,
                           CLSCTX_ALL,
                           &IID_IDirectDraw4,
                           &lpdd4)))
{
    // error
} // end if

// now before using the DirectDraw object, it must
// be initialized using the initialize method

if (FAILED(IDirectDraw4_Initialize(lpdd4,NULL)))
{
    // error
} // end if

// now that we're done with COM, uninitialize it
CoUninitialize();
```

前面的代码是微软推荐创建 DirectDraw 对象的方法。但这个技术有些不可靠，也只使用了一个宏：

```
IDirectDraw4_Initialize(lpdd4,NULL);
```

你可以去掉 (1)，使用

```
lpdd4->Initialize(NULL);
```

将其代替，使之全部成为 COM。(1)(2) 式中的 NULL 是视频设备，在本例中视频设备是系统设定的驱动器（这就是为什么要填为 NULL 的原因）。总之，不难明白前面的宏是怎样扩展为代码的。我想可能是可以变得容易些。但是，为什么微软不继续利用宏来创建新界面，如：

```
DirectDrawCreate4 (...);
```

那不是更好吗？但为什么要问为什么呢？我的观点是你应当自己这样做，这样你的程序代码看起来就会相当统一。

现在，你已经知道怎样创建一个 DirectDraw 对象，以及怎样获得最新的界面。那我们就进行下一步——设置协作等级。

和 Windows 协同工作

众所周知，Windows 是一个协作、共享的环境。虽然作为一个程序员，我至今还没有想到怎样使得我的程序与 Windows 协同工作，但这至少是一种观念。无论如何，DirectX 和任何 Win32 系统相似。最低限度，DirectX 准备调用不同资源时必须通知 Windows。这样 Windows 其他的程序就不会请求调用 DirectX 所控制使用的资源。基本上，DirectX 是一个完整的资源体，Windows 知道它在做些什么。

在 DirectDraw 例子中，你惟一需要感兴趣的是视频显示硬件。下面两个模式值得注意：

- 全屏模式
- 窗体模式

在全屏模式中，DirectDraw 的运行很像旧的 DOS 程序。这就是说，DirectDraw 给你的游戏分配了全屏画面，你可以直接对视频硬件进行操作。没有其他的应用程序能够接触到硬件。窗体模式有点不一样。在窗体模式中，DirectDraw 更多的与 Windows 协作，因为其他的应用程序可能需要更新自己的客户窗体区域（对使用者是可见的）。因此，在窗体模式中，你对视频硬件的控制和独占受到了限制。然而，你仍然有足够的权限使用 2D 和 3D 加速，这是好事。但另一方面，首先会是上窄下宽……

第 7 章“高级 DirectDraw 和位图图形”将更多的介绍窗体化的 DirectX 程序，但这更难以掌握。本章的大部分是处理全屏模式，因为这更容易，请牢记这一点。

现在你对 DirectX 和 Windows 协同工作有了一些了解，我们来看看怎样使 Windows 知道你如何与 Windows 协同工作——需要设置 DirectDraw 的协作等级，需用到 IDirectDraw4::SetCooperativeLevel() 函数（这是 IDirectDraw4 的方法）。

C++

对于 C 程序员来说，语法 IDirectDraw4::SetCooperativeLevel() 的意思可能有些模糊。:: 操作符被称为域解析操作符，这语法仅仅表示 SetCooperativeLevel() 是 IDirectDraw4 界面的方法（或成员函数）。这基本上是一个包含了数据和虚拟函数表的类。在一些情况下，我可能将不用界面作为方法的前缀，如把函数直接写作 SetCooperativeLevel()。然而，考虑到所有的 DirectX 函数是界面的一部分，因此必须使用函数指针的形式对函数进行调用，如 `lpdd->function(...)`。

下面是 IDirectDraw4::SetCooperativeLevel() 函数的原型：

```
HRESULT SetCooperativeLevel(HWND hWnd, // window handle
                             DWORD dwFlags); // control flags
```

如果执行成功的话，函数返回DD_OK，否则返回错误代码。

很有趣的是，这是在 DirectX 式中第一次出现窗体句柄，有了 hWnd 参数，DirectX（或更具体一点，DirectDraw）就指向具体的窗体。在所有的情况下，你只需使用主窗体句柄。

SetCooperativeLevel()函数的第二个参数是 dwFlags，这是控制标志参数，直接影响 DirectDraw 和 Windows 协同工作的方式。表 6.2 列出了能够通过逻辑“或”得到想要的协同等级的最常用的值。

表 6.2 SetCooperativeLevel()的控制标志

值	说 明
DDSCCL_ALLOWMODEX	允许使用Mode X (320×200, 240, 400) 显示模式。只有当 DDSCCL_EXCLUSIVE和DDSCCL_FULLSCREEN标志存在的时候才能使用
DDSCCL_ALLOWREBOOT	当处于独占（全屏）模式时，允许Ctrl+Alt+Del被检测到
DDSCCL_EXCLUSIVE	请求独占级别，该标志必须和DDSCCL_FULLSCREEN一起使用
DDSCCL_FPUSETUP	表示被调用的应用程序可能持有设置的FPU使得Direct3D性能保持最优（单精度和异常失效），所以 Direct3D不需要每次都设置FPU。至于具体内容请参看 DirectX SDK 中的“DirectDraw Cooperative Level and FPU”
DDSCCL_FULLSCREEN	表示需要全屏模式。其他程序中的GDI将不允许在屏幕上画图。这个标志必须和 DDSCCL_EXCLUSIVE一起使用
DDSCCL_MULTITHREADED	请求多线程安全DirectDraw行为。现在暂不考虑该内容
DDSCCL_NORMAL	表示应用程序将是一个规则的Windows应用程序。该标志不能和 DDSCCL_ALLOWMODEX、DDSCCL_EXCLUSIVE、DDSCCL_FULLSCREEN一起使用
DDSCCL_NOWINDOWCHANGES	表示在激活状态下，不允许DirectDraw最小化或恢复窗体

上述标志看上去有些冗余。基本上讲，DDSCCL_FULLSCREEN和DDSCCL_EXCLUSIVE必须一起使用，如果想使用任何Mode X模式，必须同时使用DDSCCL_FULLSCREEN、DDSCCL_EXCLUSIVE及DDSCCL_ALLOWMODEX。除此之外，这些标志的作用与其定义很吻合，在大多数情况下，编写全屏程序如下：

```
lpdd4->SetCooperativeLevel(hWnd,
    DDSCCL_FULLSCREEN |
    DDSCCL_ALLOWMODEX |
    DDSCCL_EXCLUSIVE |
    DDSCCL_ALLOWREBOOT);
```

普通的窗体程序如下：

```
lpdd4->SetCooperativeLevel(hWnd, DDSCCL_NORMAL);
```

当然，在本书的后面部分你将学到多线程编程技术，你可能会想加上多线程标志

DDSCL_MULTITHREAD, 使得程序更为安全。无论如何, 我们先来看看如何同时创建一个 DirectDraw 对象和设置协同等级:

```
LPDIRECTDRAW lpdd=NULL;    // standard DirectDraw 1.0
LPDIRECTDRAW lpdd4=NULL;   // DirectDraw 6.0 interface 4

// first create base IDirectDraw interface
if (FAILED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
    // error
} // end if

// now query for IDirectDraw4
if(FAILED(lpdd->QueryInterface(IID_IDirectDraw4,
                              (LPVOID *)&lpdd4)))
{
    // error
} // end if

// now set the cooperation level for windowed directdraw
// since we aren't going to do any drawing yet
if ( FAILED(lpdd4->SetCooperativeLevel(hwnd,DDSCL_NORMAL)))
{
    // error
} // end if
```

注意

为了节省版面, 省去了错误处理调用的 FAILED() 和 (或) SUCCEEDED(), 但是, 要记住你应该时刻检查错误。

现在, 你已经学会了怎样编写创建窗体的完整的 DirectX 应用程序, 启动 DirectDraw, 设置协同等级。虽然, 你还不知道怎样画图, 但这只是开始。看看 CD 中 DEMO6_1.CPP 及执行文件 DEMO6_1.EXE。当你运行程序时, 你将看到图 6.5 所示的界面。这个程序是基于 T3D Game Console 模板的。所以, 在 Game_Init() 和 Game_Shutdown() 中的惟一改变就是为 DirectDraw 创建和设置了协同等级。



图 6.5 DEMO6_1.EXE 运行界面

下面这些函数添加了一些 DEMO6_1.CPP 中的 DirectDraw 代码，所以从中你可以看出怎样建立一个简单的 DirectDraw：

```
int Game_Init(void *parms = NULL, int num_parms = 0)
{
    // this is called once after the initial windows is created and
    // before the main event loop is entered , do all your initialization
    // here

    // first create base IDirectDraw interface
    if (FAILED(DirectDrawCreate(NULL, &lpdd, NULL)))
    {
        // error
        return(0);
    } // end if

    // now query for IDirectDraw4
    if (FAILED (lpdd->QueryInterface(IID_IDirectDraw4,
                                     (LPVOID *)&lpdd4)))
    {
        // error
        return(0);
    } // end if

    // set cooperation to normal since this will be a windowed app
    lpdd4->SetCooperativeLevel(main_window_handle, DDSCL_NORMAL);

    // return success or failure or your own return code here
    return(1);

} // end Game_Init

////////////////////////////////////

int Game_Shutdown(void *parms=NULL, int num_parms=0)
{
    // this is called after the game is exited and the main event
    // loop while is exited ,do all you cleanup and shutdown here

    // simply blow away the IDirectDraw4 interface
    if (lpdd4)
    {
        lpdd4->Release();
        lpdd4=NULL;
    } // end if

    // return success or failure or your own return code here
    return(1);
}
```

```
} // end Game_Shutdown
```

提 示



如果打算跳过这些,并试图编译DEMO6_1.CPP,请记住手工从DirectX 6.0 SDK LIB\目录中添加DDRAW.LIB,同时为编译程序的.H(头文件)查找目录加上DirectX 标题路径作为第一个目录。当然,你需要生成一个Win32.EXE。我每天至少收到10封忘记加上.LIB文件的初学者发来的email,所以希望你不要重蹈覆辙。

进入事件模式

创建 DirectDraw 的下一个步骤可能是所有过程中最酷的。通常,对于基本 ROM BIOS 模式,在 DOS 下设置视频模式是相当合适的。但在 Windows 下,由于模式转换的影响,在 DOS 下设置视频模式几乎是不可能的。然而,对于 DirectX 来说,它是一个例外。DirectDraw 的主要目的之一就是使得视频模式转换对于程序员来说是直接和透明的。不再需要 VGA/CRT 控制寄存器编程,只要做一次调用就行了。Presto 这种模式可以设置成任何你想要的模式(当然是在卡支持的情况下)。

设置视频模式的函数是 SetVideoMode(),这也是 IDirectDraw4 界面的一个方法。或在 C++中,为 IDirectDraw4::SetVideoMode()。下面是它的函数原型:

```
HRESULT SetDisplayMode(DWORD dwWidth, // width of mode in pixels
                        DWORD dwHeight, // height if mode in pixels
                        DWORD dwBPP,   // bits per pixel,8,16,24,etc.
                        DWORD dwRefreshRate, // desired refresh,0 for default
                        DWORD dwFlags);      // extra flags(advanced )0 for default
```

通常,如果执行成功,函数返回DD_OK。

你可能要说:“这么好,不太可能吧!”。你曾试图建立一个 Mode X 模式,比如 320×400 或 800×600 模式吗?即使你成功了,也只是运气比较好。用这些 DirectDraw 函数,你只是传递宽度、长度、色深及用于基本存取的方法!如果能够建立请求模式,DirectDraw 就会处理所有插入式视频卡的特性。而且,这种模式保证具有线性的存储缓冲器……更多的具体内容在后面。表 6.3 列出了最通用的视频模式及其色深。

表 6.3 常用视频模式

宽 度	长 度	BPP	Mode X
320	200	8	*
320	240	8	*
320	400	8	*

续表

宽 度	长 度	BPP	Mode X
512	512	8, 16, 24, 32	
640	480	8, 16, 24, 32	
800	640	8, 16, 24, 32	
1024	768	8, 16, 24, 32	
1280	1024	8, 16, 24, 32	

有趣的是，你能够申请到任何你希望得到的模式。比如，你可以选择 400×400 ，如果视频驱动能够建立它，这个模式就会正常工作。然而，最好选用表 6.3 列出的模式，因为这些模式最为通用。

技 巧



实际上，前面我用了一个 Win32 API 函数设置视频模式，但这个函数损害了系统，并把事件弄得混乱。

回头看一下这个函数，前 3 个参数很简单，但后两个需要解释一下。`dwRefreshRate` 用于覆盖视频驱动器的默认刷新值。因此，如果你申请一个 320×320 模式，大概刷新频率是在 70Hz。对于这个参数，如果你愿意，可以把这个参数强制设置为 60Hz。但通常刷新频率我不作专门设置，只是把 `bit` 设置成 0（指示驱动程序使用默认值）。

最后一个参数 `dwFlags` 是附加标志 `WORD`，它是个垃圾箱，没有什么太大的用处。目前，这个参数用作替换值，所以你可以通过标志 `DDSDM_STANDARDVGAMODE` 使用 320×200 的 VGA 模式 13h 而不使用 Mode X 320×200 。我并不担心这个参数。如果你想用 320×320 写个游戏，你可以用这个参数实验一下，看用 VGA 模式 13h 快还是 320×200 下的 Mode X 模式快。但这个差异完全可以忽略。现在把 `dwFlags` 设置为 0。

有了足够的基础知识，现在我们开始设置转换模式！要转换模式，你必须创建 `DirectDraw` 对象，设置协同等级，最后设置显示模式，如下：

```
lpdd4->SetDisplayMode(width,height,bpp,0,0);
```

例如，如果想要在 256（8 位）色中创建一个 640×480 模式：

```
lpdd4->SetDisplayMode(640,480,8,0,0);
```

如果用 16 位色设置一个 800×600 模式：

```
lpdd4->SetDisplayMode(800,600,16,0,0);
```

现在，这两种模式已经有了相当大的区别，区别不仅仅在分辨率即色深上。8 位模式工作方式完全不同于 16 位、24 位或 32 位模式的工作方式。回想一下在前面章节 Win32/GUI 的编程中有关调色板的主题（第 3 章“高级 Windows 编程”，第 4 章“Windows GDI、控制

和突发奇想”),其有关理论也同样适用于 DirectDraw。这就是说,当你创建一个 8 位模式时,就要请求一个调色板模式,也必须创建一个调色板,并用 8.8.8 RGB 条目填充它。

另一方面,如果你创建一个直接的 RGB 模式如 16、24 或 32 位/像素,你就不需要进行这一步。(当你学会怎样做)你可以直接对视频缓冲器写入编码。至少你要学会怎样使用 DirectDraw 调色板进行工作(这是下一个讨论主题)。在继续学习新内容时,我们来看一个完整的例子,这个例子是创建一个分辨率为 $640 \times 480 \times 8$ 的全屏 DirectX 应用程序。

CD 中的 DEMO6_2.CPP 及相关可执行文件正是这样做的。我们将会看到一幅画面——一个黑色矩形。因为演示的是全屏应用软件。但可以肯定的是我们可以使用代码来使之改变。和往常一样,我们在游戏控制台程序的基础上稍加修改并将与 DirectX 相关的部分改写成 Game_Init()及 Game_Shutdown(),其代码如下所示。看了这些代码,你一定会惊讶于它们的简单。

```
int Game_Init(void *parms=NULL,int num_params=0)
{
    // this is called once after the initial window is created and
    // before the main event loop is entered, do all your initialization
    //here

    // first create base IDirectDraw interface
    if(FAILED(DirectDrawCreate(NULL, &lpdd, NULL)))
    {
        // error
        return(0);
    } // end if

    //now query for IDirectDraw4
    if (FAILED(lpdd->QueryInterface(IDD_IDirectDraw4,
                                    (LPVOID*) &lpdd4)))
    {
        // error
        return(0);
    } // end if

    // set cooperation to full screen
    if (FAILED(lpdd4->SetCooperativeLevel(main_window_handle,
        DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX |
        DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)))
    {
        // error
        return(0);
    } // end if

    //set display mod to 640x480x8
```

```

if (FAILED(lpdd4->SetDisplayMode(SCREEN_WIDTH,
                                SCREEN_HEIGHT, SCREEN_BPP, 0, 0)))
{
    //error
    return(0);
} // end if

// return success or failure or your own return code here
return(1);

} // end Game_Init

////////////////////////////////////

int Game_Shutdown(void *parms = NULL, int num_parms = 0)
{
    // this is called after the game is exited and the main event
    // loop while is exited, do all your cleanup and shutdown here

    // simply blow away the IDirectDraw4 interface
    if (lpdd4)
    {
        lpdd4->Release();
        lpdd4 = NULL;
    } // end if

    // return success or failure or your own return code here
    return(1);

} // end Game_Shutdown

```

在此，略去了控制调色板（256 色模式）和存取显示缓冲器。下面先讨论色彩问题。

巧妙的色彩

DirectDraw 支持许多不同的色深，包括 1、2、4、8、16、24 和 32 位/像素。显然，1、2 和 4 位/像素已有些过时了。因此，不必关心这些色深。另一方面，8、16、24 和 32 位/像素模式是最具有意义的。在你编写的绝大多数游戏中，可能会由于考虑到速度的原因而选择 8 位的调色板模式，或 16 位和 24 位的完全 RGB 颜色模式。RGB 模式通过往帧缓冲区里写入相同大小的字节而工作，如图 6.6 所示。调色板模式通过使用一个查询表来运作，这个表是按照帧缓冲区里独立的、单字节的像素值来检索的。因此，有 256 个不同的值，你可能以前已经见过了，所以它看起来应该很熟悉。

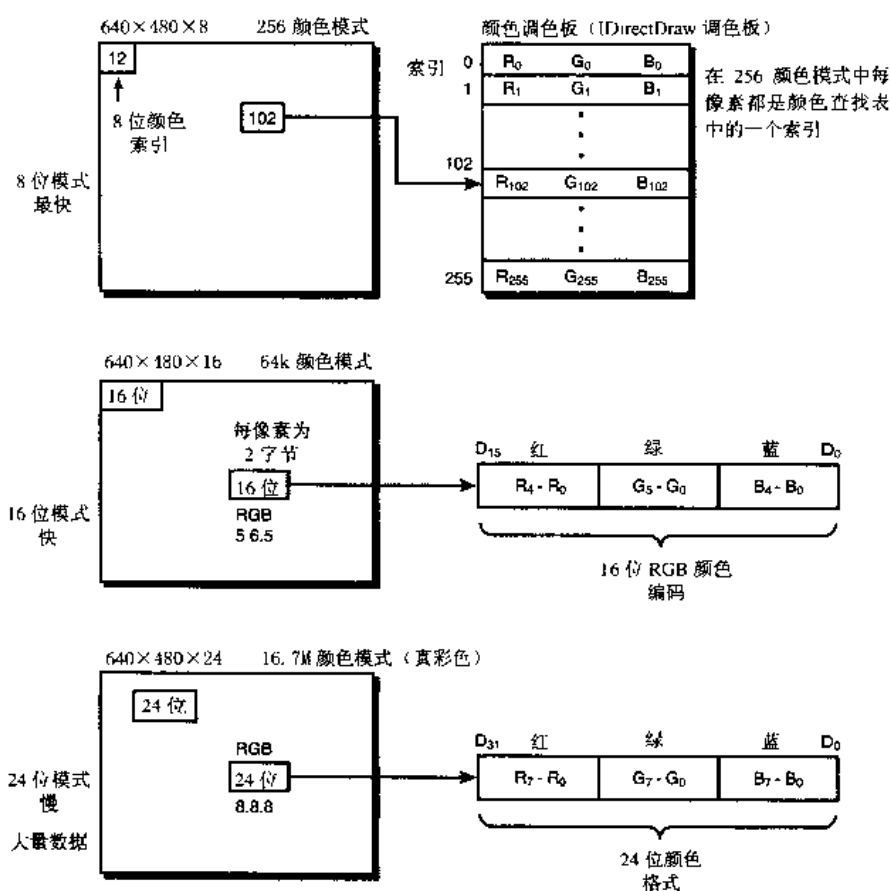


图 6.6 不同色深的对比

你需要学习如何创建一个 256 色的调色板，然后通知 DirectDraw 你想使用它。现在我们来看看具体步骤吧：

1. 创建一个或多个调色板数据结构，作为 256 色调色板的数组。
2. 从 DirectDraw 对象上创建一个调色板对象即 IDirectDrawPalette 对象。在很多情况下，这个对象将会映射到硬件 VGA 调色板寄存器中。
3. 把调色板对象附在画图表面上，所以对象中所有的数据都以适当的颜色表现出来。
4. (可选) 可以按照你的意愿改变调色板的条目或整个调色板。如果你在第二步中把调色板设置为 NULL，省去了第一步，那么你需要执行这一步。基本上，我想要说的是当你创建一个调色板界面时，你同样能够给这个界面一个色彩调色板。如果现在不想这样做，也可以以后做。因此，以后如果你还记得填充调色板条目，那可以用步骤 2 作为步骤 1。

我们开始创建调色板数据结构。调色板数据结构是基于 PALETTEENTRY Win32 结构的 256 色调色板条目的数组。其原型如下：

```
typedef struct tagPALETTEENTRY
```

```

{
    BYTE    peRed;    //    red component 8-bits
    BYTE    peGreen;  //    green component 8-bits
    BYTE    peBlue;   //    blue component 8-bits
    BYTE    peFlags;  //    control flags; set to PC_NOCOLLAPSE
} PALETTEENTRY;

```

看起来很熟悉？很好！不管怎样，创建一个调色板，你只须建立一个如下结构的数组：

```
PALETTEENTRY    palette[256];
```

然后用你想要的方式填充它。然而，这有一个规则：你必须把 `peFlags` 设置为 `PC_NOCOLLAPSE`。这是必须的，因为你不希望 Win32/DirectX 为你优化调色板。牢记这一点，这有一个创建随机调色板的例子，例子中创建的调色板，黑色在位置 0，白色在位置 255：

```

PALETTEENTRY    palette[256] ;    // palette storage

// fill em up with color!
for (int color=1; color < 255; color++)
{
    // fill with random RGB values
    palette[color].peRed    = rand()%256;
    palette[color].peGreen  = rand()%256;
    palette[color].peBlue   = rand()%256;

    // set flags field to PC_NOCOLLAPSE
    palette[color].peFlags = PC_NOCOLLAPSE;
} // end for color
// now fill in entry 0 and 255 with black and white
palette[0].peRed    = 0;
palette[0].peGreen  = 0;
palette[0].peBlue   = 0;
palette[0].peFlags  = PC_NOCOLLAPSE;

palette[255].peRed    = 255;
palette[255].peGreen  = 255;
palette[255].peBlue   = 255;
palette[255].peFlags  = PC_NOCOLLAPSE;

```

这就是调色板的全部。当然，你可以创建多个调色板并且用你所要的东西填充，那就是你个人的喜好了。

下一步就是创建实际的 `IDirectDrawPalette` 界面。幸运的是，界面并没有像 DirectX 6.0 那样有所改变，因此不需要使用 `QueryInterface()` 或其他。这里有 `IDirectDraw4::CreatePalette()` 的原型，它创建一个调色板对象：

```

HRESULT CreatePalette(DWORD dwFlags,          // control flags
LPDPALETTEENTRY lpDDColorArray,              //palette data or NULL,
LPDIRECTDRAWPALETTE FAR *lplpDDPalette,     //received palette interface
IUnknown FAR *pUnkOuter);                  //advanced, make NULL

```

如果该程序调用成功则返回 DD_OK。

让我们看看这些参数。第一个参数是 dwFlags，它控制调色板的不同特性，过一会儿还有进一步介绍。再一个参数是一个指向初始调色板的指针，如果你不想传送它就将其赋值为 NULL。再下一个是实际的 IDirectDrawPalette 界面的存储指针，如果函数调用成功它可以接收界面。最后一个是用干高级的 COM 部分，所以简单的传送 NULL 就可以了。这一串参数中最使人感兴趣的当然是 dwFlags。让我们详细地看一下你可能的选择。参考表 6.4 选择可能的值来创建标志。

表 6.4 CreatePalette() 的控制标志

值	说 明
DDPCAPS_1BIT	1位颜色。颜色表中有2条
DDPCAPS_2BIT	2位颜色。颜色表中有4条
DDPCAPS_3BIT	4位颜色。颜色表中有16条
DDPCAPS_4BIT	8位颜色。最为普遍。颜色表中有256条
DDPCAPS_BBITENTRIES	这是一个被称为索引化调色板的高级特征。适合于1、2、4位调色板。只要不用就行了
DDPCAPS_ALPHA	与PALETTEENTRY相关结构的peFlags成员被解释为控制透明度的单8位alpha值。用该标志创建的调色板只能连接到被DDSCAPS_TEXTURE创建的画面中。同样，这是用于高级部分
DDPCAPS_ALLOW256	表明这个调色板总共含有256条定义。通常，0和255分别代表黑色与白色，并且在NT系统下任何时候你不能写这两个条目。然而，在大多数情况下，你不需要该标志，因为0无论如何都是黑色，大多数调色板把255作为白色。可根据你自己的喜好来做
DDPCAPS_INITIALIZE	使用lpDDColorArray传递的颜色数组中的颜色来初始化调色板。这就使调色板中的数据传送到被下载的硬件调色板中
DDPCAPS_PRIMARYSURFACE	调色板与主画面连接。除非DDPSETPAL_VSYNC被说明而且支持它，否则只要调色板中的颜色表一改变就会影响显示
DDPCAPS_VSYNC	在纵向空白期间强迫调色板适时更新，这样来最小化非规则颜色及将之分离。然而这一点尚未被完全支持

如果你问我的话，我会告诉你有许多令人迷惑的控制字。最基本的，你只需用 8 位调色板工作，需要进行 OR 运算的控制标志是：

DDPCAPS_BBIT | DDPCAPS_ALLOW256 | DDPCAPS_INITIALIZE

如果你并不关心颜色 0 和颜色 255，你就可以省略 DDPCAPS_ALLOW256。而且如果在调用 CreateaPalette()时你不打算传送调色板，那么你就可以省略 DDPCAPS_INITIALIZE。

掌握了上面内容，你就可以使用随机调色板创建一个调色板对象：

```

LPSIRECTDRAWPALETTE lpddpal=NULL; //palette interface
if(FAILED(lpdd4->CreatPalette(DDPCAPS_256 |
                             DDPCAPS_ALLOW256 |
                             DDPCAPS_INITIALIZE,
                             palette,
                             &lpddal,
                             NULL)))
{
    //error

} // end if

```

如果程序执行成功，`lpddpal` 将返回一个有效的 `IDirectDrawPalette`。而且计算机硬件使用传递的调色板即时更新色彩调色板，在这个例子中则是随机使用 256 色。

关于这一点我想给出一个演示，但不幸的是在 `DirectDraw` 中我们陷入了那所谓“鸡和蛋”的矛盾之中，那就是直到可以在屏幕上画图时你才能看见颜色。所以下面要讲这问题。

创建一个显示画面

正如你所知，屏幕上所显示的图像只不过是内存中某种格式、调色板或者 RGB 彩色像素矩阵的表现。不管用哪种方式，都可以得到相同的效果，你所需要知道的是怎样把数据写入内存。然而，在 `DirectDraw` 下设计者决定略微抽象视屏内存的概念，以达到无论系统中的视屏卡是何种类型，存取视屏画面的方法都是相同的（程序员的视图观点）。如此一来，`DirectDraw` 就支持各种画面。

参考图 6.7，这些画面都是能够记录位图数据的、保存在内存中的矩形区域。并且有两种画面：主画面和辅助画面。主画面直接对应被视屏卡更新的实际视屏内存，而且始终可以看到。因此，在 `DirectDraw` 程序中，你将仅仅拥有一个主画面，它直接指向屏幕图像并且常驻在 VRAM 中。当你控制它时，你就会看到及时更新的屏幕结果。如果你设定视屏模式为 $640 \times 480 \times 256$ ，那么你也需要把主画面设为 $640 \times 480 \times 256$ ，而且要把它粘贴到显示装置——`IDirectDraw4` 对象上。

换句话说，辅助画面更灵活。它们可以是任意大小，可以驻于 VRAM 中或系统内存中，在内存允许的范围内你可以尽可能多地创建辅助画面。在大多数情况下，为获得生动平滑的主画面，你需要创建一到两个辅助画面作为缓冲。这些画面将始终拥有同主画面一样的颜色深度和几何形状，并且使用动画的下一帧进行刷新。随后为获得生动平滑的主画面，这些辅助画面被迅速地拷贝或交换到主画面。这就是所谓的双倍或三倍缓冲。在下面的章节中你将了解更多的有关辅助画面的使用方面的内容。

辅助画面的第二个用途是用于控制游戏中代表对象的位图图像和动画。这是 `DirectDraw` 一个非常重要的特征，因为只有使用 `DirectDraw` 的画面，你才可以调用硬件加速器处理位

图数据。如果你使用自己编写的位图图像转换软件来处理位图，那就损失了硬件加速。

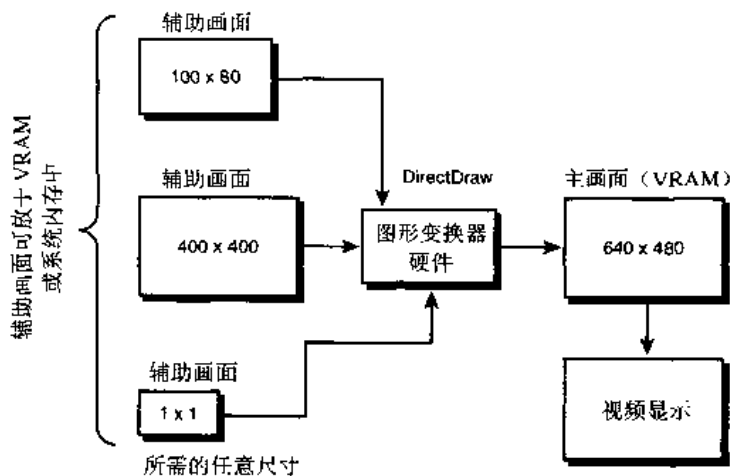


图 6.7 画面可为任意尺寸

现在让我们看看如何创建一个大小同你的显示模式一致的主画面，然后你将学会向其中写入数据及在屏幕上绘制像素。

创建一个主画面

不管创建什么画面，你都必须遵循下面的步骤：

1. 填写一个DDSURFACEDESC2数据结构，它将描述你想创建的画面。
2. 调用IDirectDraw4::CreateSurface()来创建画面。

下面是CreateSurface()的原型：

```
HRESULT CreateSurface(
    LPDDSURFACEDESC2 lpDDSurfaceDesc2,
    LPDIRECTDRAWSURFACE4 FAR *;lp1DDSurface,
    Iunknown FAR *pUnkOuter);
```

程序接收一个所要创建 DirectDraw 界面的描述，使用一个指针来获取界面，最后，将 NULL 赋给 COM 的高级属性 pUnkOuter。填写这一数据结构的过程有一些令人迷惑，但我会一步一步地帮助你。首先，让我们看一看 DDSURFACEDESC2：

```
typedef struct _DDSRUFACEDESC2
{
    DWORD dwSize;          //size of this structure
    DWORD dwFlags;         //control flags
    DWORD dwHeight;        //height of surface in pixels
    DWORD dwWidth;         //width of surface in pixels
    Union
```

```

{
    LONG lpitch;           //memory pitch per row
    DWORD dwLinearSize;    //size of the buffer in bytes
} DUMMYUNIONNAMEN(1);
    DWORD dwBackBufferCount; //number of back buffers chained
    Union
    {
        DWORD dwMipMapCount;           //number of mip-map levels
        DWORD dwRefreshRate;           //refresh rate
    } DUMMYUNIONNAMEN(2);
    DWORD dwAlphaBitDepth;           //number of alpha bits
    DWORD dwReserved;               //reserved
    Lpvoid lpSurface;                //pointer to surface memory
    DDSCOLORKEY ddckCKDestOverlay;   //dest overlay color key
    DDSCOLORKEY ddckCKDestBlit;      //destination color key
    DDSCOLORKEY ddckCKSrcOverlay;    //source overlay color key
    DDSCOLORKEY ddckCKSrcBlit;       //source color key
    DDPIXELFORMAT ddtpixelFormat;    //pixel format of surface
    DDSCAPS2 ddsCaps;                //surface capabilities
    DWORD dwTextureStage;             //used to bind a texture
                                        //to specific stage of D3D
} DDSURFACEDESC2, FAR* LPDDSURFACEDESC2;

```

正如你所看到的，这是一个复杂的结构。而且 75% 的参数含义相当模糊。但幸运的是你只需知道我用黑体标出的部分。让我们看看它们各自的详细功能：

dwSize——这是所有 DirectX 数据结构中最为重要的参数之一。许多 DirectX 数据结构通过地址传送，因此接收程序或方法并不知道数据结构的大小。相反，如果第一个 32 位值总是代表数据结构的大小的话，那么接收程序就会通过提取第一个参数 **DWORD** 知道有多少数据量。因此，DirectDraw 和 DirectX 数据结构通常把所有结构的第一个元素作为数据量说明。这一点看起来似乎很多余，但这样设计的确非常好用。所有你需要做的事情就是按照下面格式进行参数化：

```

DDSURFACEDESC2 ddsd;
ddsd.dwSize=sizeof(DDSURFACEDESC2);

```

dwFlags——这个字段用来指出 DirectDraw 中需要使用有效值填充的参数，或是在查询操作中使用该结构时需要获得参数的字段。看一看表 6.5，该表中列出了标志字所有可能的取值。比如要将有效值放入 **dwWidth** 和 **dwHeight** 字段，你就要这样设定 **dwFlags**：

```

ddsd.dwFlags=DDSD_WIDTH | DDSD_HEIGHT;

```


然后, DirectDraw 就会查找 dwHeight 和 dwWidth 的值, 并查看数据是否有效。同时将 dwFlags 作为有效数据的说明符。

表 6.5 DDSURFACEDESC2 中 dwFlags 的不同取值

值	说 明
DDSD_ALPHABITDEPTH	表示dwAlphaBitDepth成员有效
DDSD_BACKBUFFERCOUNT	表示dwBackBufferCount成员有效
DDSD_CAPS	表示ddsCaps成员有效
DDSD_CKDESTBLT	表示ddckCKDestBlit成员有效
DDSD_CKDESTOVERLAY	表示ddckCKDestOverlay成员有效
DDSD_CKSRCBLT	表示ddckCKSrcBlit成员有效
DDSD_CKSRCOVERLAY	表示ddckCKSrcOverlay成员有效
DDSD_HEIGHT	表示dwHeight成员有效
DDSD_LINEARIZE	表示 dwLinearSize成员有效
DDSD_LPSURFACE	表示lpSurface成员有效
DDSD_MIPMAPCOUNT	表示dwMipMapCount成员有效
DDSD_PITCH	表示lPitch成员有效
DDSD_PIXELFORMAT	表示ddpfPixelFormat成员有效
DDSD_REFRESHRATE	表示dwRefreshRate成员有效
DDSD_TEXTURESTAGE	表示dwTextureStage成员有效
DDSD_WIDTH	表示dwWidth成员有效

dwWidth——表示像素中的画面宽度。当你创建一个画面时这就是设定宽度的地方, 如320, 640等。而且, 如果查询画面的特性, 这一参数将返回画面的宽度(如果你需要)。

dwHeight——表示像素中的画面高度。同dwWidth相似, 这是你创建画面时设定画面高度的地方, 如200, 240, 480等。

lPitch——这是一个有趣的参数。首先它是你所设定的演示模式的水平内存间距。参考一下图6.8, lPitch是视频模式中每行的字节数, 当然也可以视为内存跨度或宽度。当然这也是非常重要的数据块, 因为: 当你需要视频模式为 $640 \times 480 \times 8$ 时, 你就知道每行有640个像素, 每个像素由8位(一个字节)组成。所以, 确切说每行由640个字节组成, 这样lPitch应该是640, 对吗? 实际上不一定。

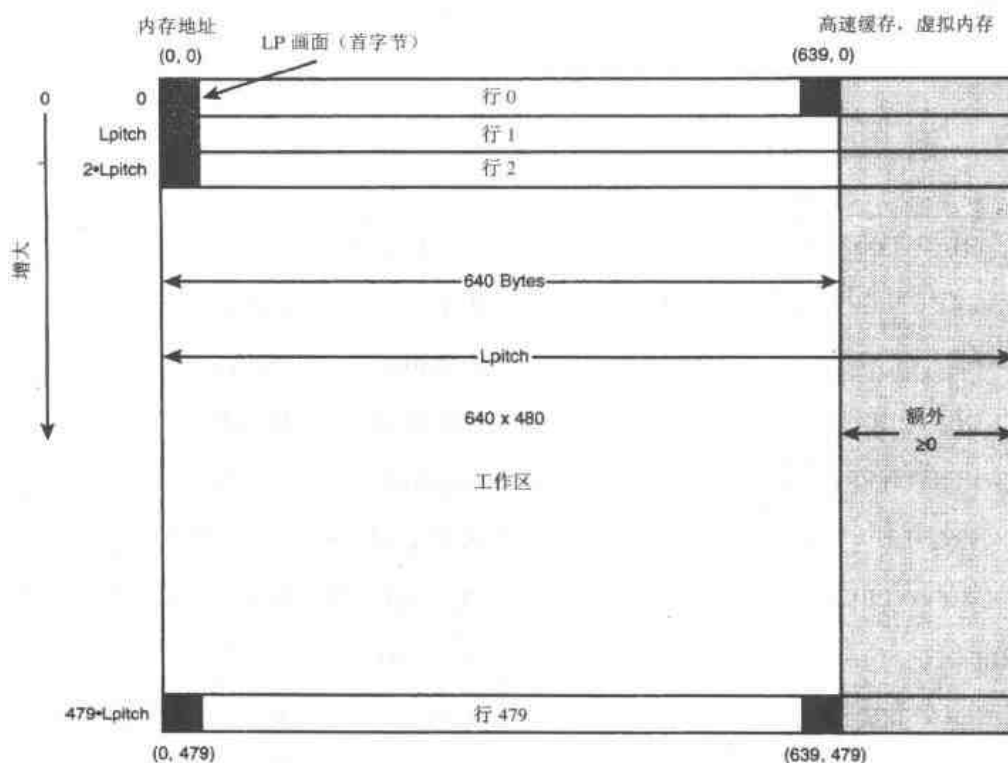


图 6.8 进入主画面

大多数视频板支持线性内存模式，而且拥有寻址硬件，因此这一属性要取为 `true`，但却并不保证总是这样。所以你不能假定一个 $640 \times 480 \times 8$ 的视频模式就是每行 640 字节。这就是 `LPitch` 的用途了。你必须参考它来确保你的内存地址计算正确，以便你可以在行之间移动。为了得到一个 $640 \times 480 \times 8$ (256 色) 模式的显示模式，你可以采用下面的代码（假定你已经要求 `DirectDraw` 将 `LPitch` 和 `lpSurface` 指向画面内存）：

```
ddsd.lpSurface[x+y*ddsd.lPitch]=color;
```

很简单，不是吗？在大多数情况下，对于 $640 \times 480 \times 8$ 模式 `ddsd.LPitch` 应该是 640，对于 $640 \times 480 \times 16$ 模式 `ddsd.LPitch` 应该是 1280（每个像素两个字节）。但是对于一些板卡来说，由于显卡上的内存寻址方式不尽相同，这一点不是一成不变的。无论是因为卡上的内部缓存还是其他原因，始终使用 `LPitch` 来进行内存计算，总是安全的。

技巧

即使 `LPitch` 不能等同于你所设定模式的横向结果，它也值得检测，以利于切换到更优化的函数。例如在初始化你的代码时，你可以得到 `LPitch` 的值及同所挑选的横向结果进行对比。如果它们是相同的，你就可以选择更理想的代码，其中肯定包含了每行的字节数。

lpSurface——这个参数是用来重新取得指向画面所在的内存的指针。内存可能是在 VRAM 或系统内存中。但你并不需要考虑它。一旦你让指针指向它，就可以像控制其他内

存一样控制它——向它写数据，从它读数据等等。这就是像素的确切绘制过程。建立指针要费点事，但是我们一会儿就会完成。最基本的，你必须“锁定”画面的内存，并且通知DirectX你打算用它工作，没有其他进程向该内存空间读/写数据。而且，当你确实得到这个指针时，依靠颜色度——8、16、24、32位像素——你就可以引用并把它赋给一个工作的化名指针。

dwBackBufferCount——这个参数是用于设定或读取后部缓冲器的数目，或者与主画面相连的辅助分页缓冲器的数目。如果你要重新调用，后部缓冲器就会通过创建一个或多个虚拟初级缓冲（具有同样的几何与颜色特性）来实现流畅的画面，这些缓冲是在显示区外的。接下来要在后部缓冲中绘制画面，这一点使用者是看不到的。随后就是翻页或复制后部缓冲到初级缓冲来显示。如果你只有一个后部缓冲，这个技术就叫做双级缓冲。用两个后部缓冲叫做三级缓冲，这样是有好处的，但占的内存较多。为了使问题简单化，在大多数场合你需要创建可以容纳一个单一主画面和一个后部缓冲的分页链。

ddckCKDestBlt——这个参数是用于控制目标颜色关键字，用于在进行位图转换操作时控制写入的颜色。在后面的第七章将介绍“高级DirectDraw和位图图形”。

ddckCKSrcBlt——该字段用于表示原来的颜色关键字，这是在执行位图操作时不需要改变的颜色值。这就是对你的位图设定透明颜色的方法。第七章中将有更多的介绍。

ddpfPixelFormat——该字段用于重新得到画面的像素格式，如果想弄明白画面的特性的话，那么这一点是非常重要的。下面是通用的结构，但是你必须查看一下DirectX SDK中的有关细节，因为它们很长，并且不是完全相关的：

```
typedef struct _DDPIXELFORMAT
{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwFourCC;
    Union
    {
        DWORD dwRGBBitCount;
        DWORD dwYUVBitCount;
        DWORD dwZBufferBitDepth;
        DWORD dwAlphaBitDepth;
        DWORD dwLuminanceBitCount; //new for DirectX 6.0
        DWORD dwBumpBitCount; // new for DirectX 6.0
    } DUMMYUNIONNAMEN(1);
    union
    {
        DWORD dwRBitMask;
        DWORD dwYBitMask;
        DWORD dwStencilBitDepth; // new for DirectX 6.0
        DWORD dwLuminanceBitMask; // new for DirectX 6.0
        DWORD dwBumpDuBitMaqsk; // new for DirectX 6.0
    } DUMMYUNIONNAMEN(2);
}
```

```

        union
        {
            DWORD dwGBitMask;
            DWORD dwUBitMask;
            DWORD dwZBitMask;           // new for DirectX 6.0
            DWORD dwBumpDvBitMask;      // new for DirectX 6.0
        } DUMMYUNIONNAMEN(3);
        union
        {
            DWORD dwBBitMask;
            DWORD dwVBitMask;
            DWORD dwStencilBitMask;      // new for DirectX 6.0
            DWORD dwBumpLuminanceBitMask; // new for DirectX 6.0
        } DUMMYUNIONNAMEN(4);
    union
    {
        DWORD dwRGBAlphaBitMask;
        DWORD dwYUVAAlphaBitMask;
        DWORD dwLuminanceAlphaBitMask; // new for DirectX 6.0
        DWORD dwRGBZBitMask;
        DWORD dwYUVZBitMask;
    } DUMMYUNIONNAMEN(5);
} DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;

```

注 意

我用黑体标出了一些常用的参数。

ddsCaps——这个参数是用来表明在其他地方没有被定义而又需要的画面的特性。实际上，这个参数是另一种数据结构体。

DDSCAPS2的结构为：

```

typedef struct _DDSCAPS2
{
    DWORD dwCaps;      //Surface capabilities
    DWORD dwCaps2;      //More surface capabilities
    DWORD dwCaps3;      //future expansion
    DWORD dwCaps4;      //future expansion
} DDSCAPS2, FAR* LPDDSCAPS2;

```

在 99%的情况下，你只需设定第一个参数。**dwCaps**.**dwCaps2** 是用于 3D 图形，是保留域，**dwCaps3** 和 **dwCaps4** 留作未来扩展使用。表 6.6 列出了 **dwCaps** 的部分标志设定。要得到全部的列表，请参阅 DirectX SDK。

例如，当创建一个初始画面时，可以这样设定 **ddsd.ddsCaps**：

```
ddsd.ddsCaps.dwCaps=DDSCAPS_PRIMARYSURFACE;
```

我知道这看起来有些复杂,在某种程度上确实如此。掌握双重嵌套的控制标志需要做一些努力。

表 6.6 DirectDraw 画面的性能控制设定

值	说 明
DDSCAPS_BACKBUFFER	画面是分页结构的后部缓冲
DDSCAPS_COMPLEX	表明正在描述一个复杂的画面。复杂画面是由一个基画面和一个或多个用来创建分页链的后部缓冲构成
DDSCAPS_FLIP	表明该画面是一个分页结构的画面的一部分。当该性能用于方法CreateSurface()时,一个前端缓冲或一个或多个后部缓冲就产生了
DDSCAPS_LOCALVIDMEM	表明这个画面在本机内存中,而不在机外的视频内存中。如果定义了该标志,那么DDSCAPS_VIDEMOEMORY也同样要求被定义
DDSCAPS_MODEX	表明该画面是320×200或320×240模式的X画面
DDSCAPS_NONLOCALVIDMEM	表明该画面存在于非本机的视频内存中。如该标志被定义了,那么DDSCAPS_VIDEMOEMORY也必须被定义
DDSCAPS_OFFSCREENPLAIN	表明该画面是一个视频外的画面而非专门的画面,比如覆盖图、文本、Z型缓冲、前端缓冲、后部缓冲或alpha画面。常常用得很巧妙
DDSCAPS_OWNDC	表明该画面将长时间有一个图案
DDSCAPS_PRIMARYSURFACE	表明该画面是主画面。它代表着使用者时刻可以看到的部分
DDSCAPS_STANDARDVGAMODE	表明该画面是一个标准的VGA模式而非X模式画面。该标志不能同DDSCAPS_MODEX一起使用
DDSCAPS_SYSTEMMEMORY	表明该画面的内存是从系统内存中分配
DDSCAPS_VIDEMOEMORY	表明该画面存在于显存中

现在你应对 DirectDraw 的复杂而强大的创建画面功能有所了解,让我们把这些知识用于工作,创建一个尺寸和颜色与显示模式(默认)一样的主画面。下面是相应的代码:

```
// interface pointer to hold primary surface, note that
// it's the 4th revision of the interface

LPDIRECTDRAWSURFACE4 lpddsprimary=NULL;

DDSURFACEDESC2 ddsd; //the DirectDraw surface description

//MS recommends clearing out the structure
memset (&ddsd,0, sizeof(ddsd)); //could use ZeroMemory()

//now fill in size of structure
ddsd.dwSize=sizeof(ddsd);
```

```

//enable data fields with values from table 6.5 that we
//will send valid data in
//in this case only the ddsCaps field is enabled, we
//could have enabled the width, height etc., but they
//aren't needed since primary surfaces take on the
//dimensions of the display mode by default
ddsd.dwFlags=DDSD_CAPS;

//now set the capabilities that we want from table 6.6
ddsd.ddsCaps.dwCaps=DDSCAPS_PRIMARYSURFACE;

//now create the primary surface

if (FAILED(lpdd->CreateSurface(&ddsd,&lpddsprimary,NULL)))
{
    //error
} //end if

```

如果该程序执行成功，`lpddsprimary` 将指向新画面的界面，你可以调用相关的方法（有好几种，比如连接 256 色调色板）来处理它。下面让我们看一下恢复调色板的例子。

选用调色板

在以前的章节中除了连接到画面之外，有关调色板的操作我们都学习过了。在创建了调色板并用相关色彩条目填充后，因为你没有绘制一个画面，所以无法把调色板与之相关联。现在你拥有了主画面，就可以完成这一步了。

向任何画面加载调色板，你所需做的就是使用 `IDirectDrawSurface4::SetPalette()` 函数，它的原型为：

```
HUESULT SetPalette (LPDIRECTDRAWPALETTE lpDDPalette);
```

这个函数仅仅使用了一个指向你所想调入的调色板的指针。要使用以前所创建的调色板，就需要使用下面的代码来完成调色板与主画面间的通信。

```

If (FAILED(lpddsprimary->SetPalette(lpddpal)))
{
    //error
} //end if

```

不是很难，是吧？目前，你已具备了一切条件来模拟一个栩栩如生的 DOS32 游戏。你可以切换视频模式，设定调色板，创建一个主画面来体现生动的视频图像。当然，仍有一些细节需要了解，如锁定主画面内存、访问 VRAM 和绘制像素。现在让我们逐一讨论。

绘图像素

要在一个全屏的 DirectDraw 模式下绘制像素点, 首先你必须建立 DirectDraw, 设定协作等级、显示模式, 创建至少一个主画面。然后你要获得访问主画面的入口, 并且向视频内存写入数据。不过, 在学习这些以前, 让我们先看一看视频画面如何工作。

如前所述, 所有的 DirectDraw 视频模式和画面都是线性的, 如图 6.9 所示。这意味着在你从上到下, 从左到右内存是线性增加的。

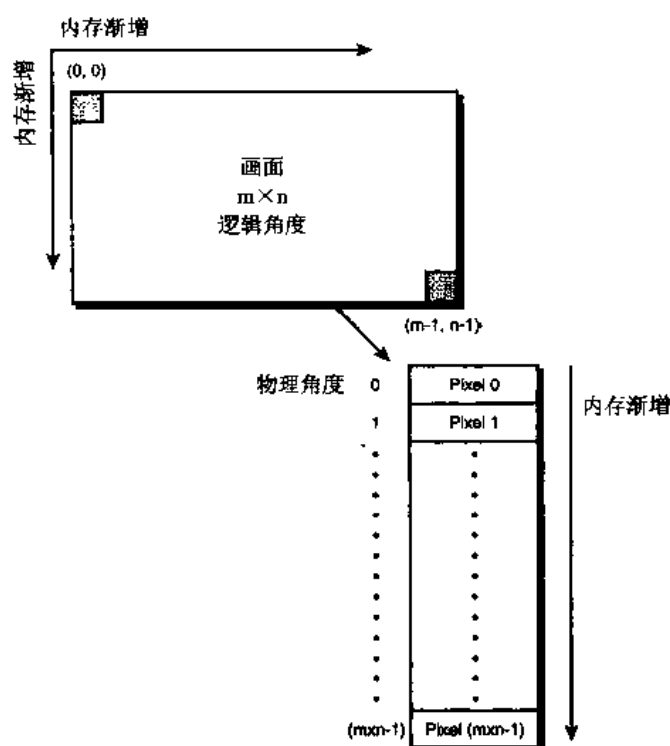


图 6.9 DirectDraw 画面是线性的

提示



也许你想知道如果视频卡不支持这一特性, DirectDraw 是如何神奇地将非线性的视频模式转换为线性模式的。比如模式 X 是完全的非线性和不可转换的。那么, 真实的情况就是——当 DirectDraw 检测到硬件中有一个非线性模式时, 一个称作 VFLATD.VXD 的驱动程序就被激活, 它会在你和 VRAM 之间建立一个软件层, 并且使 VRAM 看起来如同线性的一样。但切记这会降低速度。

另外, 要在视频缓冲中定位, 你只需要记住两条信息: 每行的内存数量和每个像素的大小 (8 位、16 位、24 位、32 位)。你可以使用下面的公式:

```
// assume this points to VRAM or the surface memory
UCHAR * video_bufferB;
Video_buffer16[x+y*(memory_pitchB>>1)]=pixel_color_16;
```

当然，这不完全正确，因为这个公式仅适用 8 位模式，或每个像素有一个 BYTE 的模式，你还需添加些如下的代码：

```
// assume this points to VRAM or the surface memory
USHORT * video_buffer16;
Video_buffer[x + y*(memory_pitchB >> 1)] = pixel_color_16;
```

这儿还有许多要说，所以让我们先仔细地看一下代码。由于我们使用了 16 位模式，我就用指针 USHORT 指向 VRAM。这样做就是为了存取队列，但是使用 16 位指针算法。因此当我假定

```
video_buffer16[1]
```

时，就确实可以访问到第二个 SHORT 或者字节对 2, 3。而且，因为 memory_pitchB 是在字节队中，必须通过向右移一位把它分成两部分，以使其位于 SHORT 或 16 位的内存间距中。最后，pixel_color16 的赋值也同样叫人费解，因为现在完整的 16 位 USHORT 将被写入视频缓冲区中，而不是像以前例子中的 8 位值。而且，8 位值将作为颜色的索引，鉴于 16 位值必须是 RGB 值，通常用 R5G6B5 的格式编码，或者说 5 位红，6 位绿，5 位蓝，如图 6.10 所示。

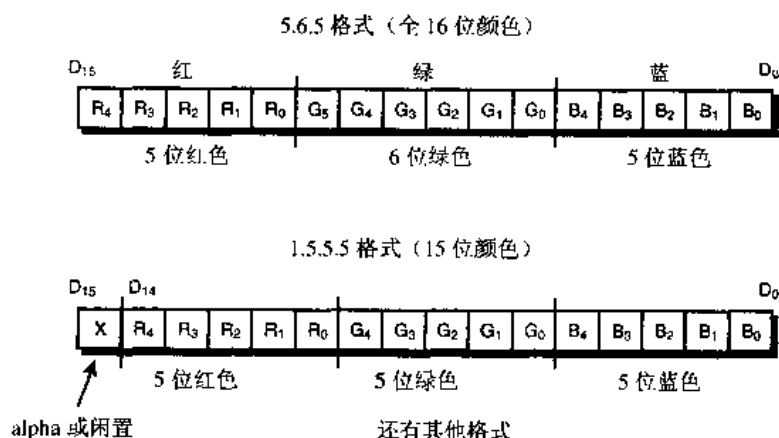


图 6.10 16 位 RGB 编码包含 5.6.5 格式

这里用一个宏来组建一个 16 位 RGB 字：

```
//this builds a 16 bit color value
#define RGB16BIT565(r,g,b) ((b<32)+(g<64)<<5)+(r<32)<<11)
```

你看到了，16 位和 RGB 模式通常比 256 色的 8 位模式在表达上和控制上更复杂一些，所以让我们就从这儿开始。要访问任何画面——主画面、辅助画面等等，你必须锁定或解除锁定显存。上锁和解锁的顺序很关键，这是因为：第一，通知 DirectDraw 你要控制内存（也就是说，不应该被其他的使用者介入），第二，指示视频硬件在你处理锁定内存时不进

行任何缓存或虚拟的内存缓冲区的操作。记住,不能确保 VRAM 待在原来的位置。但你可以锁住它,内存将待在原来的地址空间直到锁住时间完毕,在此期间你可以进行读写操作。锁住内存的函数叫做 `IDirectDrawSurface4::Lock()`,下面是它的原型:

```
HRESULT Lock(DD_RECT* lpDestRect,           //destination RECT to lock
             LPDD_SURFACE_DESC2 pDDSurfaceDesc, //address of struct to receive info
             DWORD dwFlags,                  //request flags
             HANDLE hEvent);                 //advanced,make NULL
```

这些参数并不难,但却有一些新的用途。让我们一步步来看。第一个要锁定的参数是画面内存区域的 `RECT`,如图 6.11 所示。`DirectDraw` 仅仅锁住画面内存的一部分,以便另一个进程访问没有锁定的那部分画面,进行相应的处理。这一点非常重要,因为如果你打算只更新画面的一部分,就不需要完全锁住全部画面。然而大多数情况下可以锁住整个画面以使问题简单化。通过传递 `NULL` 就可以完成。

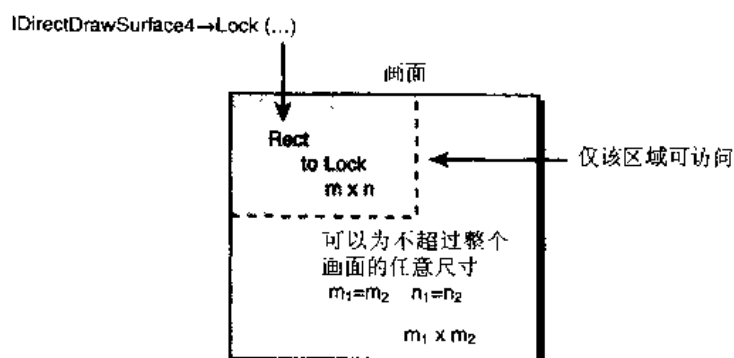


图 6.11 锁定画面的内存

第二个参数是 `DD_SURFACE_DESC2` 的地址,它需要用你所要求的画面信息来填充。最简单的就是传递一个空的 `DD_SURFACE_DESC2`。下一个参数, `dwFlags` 通知 `Lock()` 你要进行的操作。表 6.7 列出了最常用的值。

表 6.7 `Lock()` 方法的控制标志

值	说 明
<code>DDLOCK_READONLY</code>	表明画面以只读方式锁住
<code>DDLOCK_SURFACEMEMORYPTR</code>	表明应该返回一个指向 <code>RECT</code> 顶端的有效指针。如果没有矩形则返回指向画面顶端的指针。这是默认设置
<code>DDLOCK_WAIT</code>	如果在进行过程中不能加锁,该方法将重试,直到成功或产生一个如 <code>DDERR_SURFACEBUSY</code> 的错误
<code>DDLOCK_WRITEONLY</code>	表明画面是以可写的方式锁住

注 意



我用黑体标出了一些常用的标志。

最后一个参数强化了被 Win32 支持的称为“事件”的高级特性。把它设定为 NULL。

锁住基本画面确实很容易。你所想做的就是要求内存指针指向画面，同时要求 DirectDraw 等待画面成为当前画面。程序如下：

```
DDSURFACEDESC2 ddsd;           //this will hold the results of the lock

//clear the surface description out always
memset(&ddsd, 0, sizeof(ddsd));

//set the size field always
ddsd.dwsize=sizeof(ddsd) ;

//lock the surface
if (FAILED(lpddsprimary->Lock(NULL,
&ddsd,
DDLOCK_SURFACEMEMORYPTR|DDLOCK_WAIT,NULL)))
{
//ERROR
}

//***** at this point there are two fields that we are
// concerned with: ddsd.lpitch which contains the memory
//pitch in bytes per line and ddsd.lpSurface which is a
// pointer to the top left corner of the locked surface
```

一旦锁住了画面，你就可以随意地操作画面的内存。每行的存储量记录在 ddsd.lpitch 中，指向实际画面的指针记录在 ddsd.lpSurface 中。所以，如果你用的是 8 位模式（每个像素 1 个字节），下面的函数可以用于在主画面的任何地方绘制像素：

```
inline void Plot8(int x, int y,           //position of pixel
                  UCHAR color,           //color index of pixel
                  UCHAR *buffer,         //pointer to surface memory
                  Int mempitch)           //memory pitch bytes per line
{
// this function plots a single pixel
buffer[x+y*mempitch] = color;
} // end Plot8
```

下面是如何用颜色索引 26 绘制 (100, 20) 处的像素：

```
Plot8(100,20,26, (UCHAR *)ddsd.lpSurface, (int)ddsd.lpitch);
```

类似的, 下面是 16 位 5.6.5RGB 模式的绘制函数:

```
inline void Plot16(int x, int y, // position of pixel
    UCHAR red,
    UCHAR green,
    UCHAR,blue // RGB color of pixel
    USHORT *buffer, // pointer to surface memory
    int mempitch) // memory pitch bytes per line
{
    // this function plots a single pixel
    buffer[x+y*(mempitch>>1)] = _RGB168BIT565(red,green,blue);
} // end Plot16
```

这里是如何用 RGB (10, 14, 30) 在点 (300, 100) 绘制像素:

```
Plot16(300,100,10,14,30(USHORT *)ddsd.lpSurface,(int)ddsd.lPitch);
```

现在, 一旦完成当前画面帧的所有视频画面存取操作后, 你就需要解除锁定。这需要
使用 IDirectDrawSurface4::Unlock() 方法完成:

```
HRESULT Unlock(LPRECT lpRect);
```

将锁定操作使用的 RECT 发送给 Unlock()函数。如果你锁住全部画面, 则需要向该函数发送 NULL。下面是解除锁定的操作代码:

```
if (FAILED(lpddsprimary->Unlock(NULL)))
{
    //ERROR
} //end if
```

代码就是这么简单。现在, 让我们看一看在屏幕上绘制随机像素的全部步骤 (没有经过错误检查):

```
LPDIRECTDRAW lpdd=NULL; //standard DirectDraw 1.0
LPDIRECTDRAW lpdd4=NULL; //DirectDraw 6.0 interface 4
LPDIRECTDRAWSURFACE4 lpddsprimary=NULL; //surface ptr
DDSURFACEDESC2 ddsd; //surface description
LPDIRECTDRAWPALETTE lpddpal=NULL; //palette interface
PALETTEENTRY palette[256]; //palette storage

//first create base IDirectDraw interface
DirectDrawCreate(NULL,&lpdd,NULL);

//now query for IDirectDraw4
lpdd->QueryInterface(IID_IDirectDraw4,
    (LPVOID *)&lpdd4);
```

```

//release lpdd
lpdd->Release();

//set the cooperative level for full-screen mode
lpdd4->SetCooperativeLevel(hwnd,
DDSCL_FULLSCREEN |
DDSCL_ALLOWNOEX |
DDSCL_EXCLUSIVE |
DDSCL_ALLOWREBOOT);

/ set the display mode to 640x480x256
lpdd4->SetCooperativeLevel(640,480,8,0,0);

//clear ddsd and set size
memset(&ddsd,0,sizeof(ddsd));
ddsd.dwFlags=DDSD_CAPS;

//request primary surface
ddsd.ddsCaps.dwCaps=DDSCAPS_PRIMARYSURFACE;

//create the primary surface
lpdd4->CreateSurface(&ddsd,&lpddsprimary,NULL);

//build up the palette data array
for (int color =1;color<255;color++)
{
    //fill with random RGB values
    palette[color].peRed=rand()%256;
    palette[color].peGreen=rand()%256;
    palette[color].peBlue=rand()%256;
    //set flags field to PC_NOCOLLAPSE
    palette[color].peFlags=PC_NOCOLLAPSE;
}

//now fill in entry 0 and 255 with black and white
palette[0].peRed = 0;
palette[1].peGreen = 0;
palette[2].peBlue = 0;
palette[3].peFlags = PC_NOCOLLAPSE;

palette[255].peRed = 255;
palette[255].peGreen = 255;
palette[255].peBlue = 255;
palette[255].peFlags = PC_NOCOLLAPSE;

//create the palette object
lpdd4->CreatePalette(DDPCAPS_8BIT | DDPCAPS_ALLOW256 |

```

```

DDPCAPS_INITIALIZE,
Palette,&lpddpal,NULL);

//finally attach the palette to the primary surface
lpddsprimary->SetPalette(lpddpal);

//and you're ready to rock n roll;
//lock the surface first and retrieve memory pointer
//and memory pitch

//clear ddsd and set size ,never assume it's clean
memset(&ddsd, 0,sizeof(ddsd));
ddsd.dwSize=sizeof(ddsd);

lpddsprimary->Lock(NULL,&ddsd,
                DDLOCK_SURFACEMEMORYPTR |DDLOCK_WAIT,NULL))

//now ddsd.lPitch is valid and so is ddsd.lpSurface

//make a couple aliases to make code cleaner, so wie don't
//have to cast
int mempitch=ddsd.lPitch;

//plot 1000 random pixels with random colors on the
//primary surface, they will be instantly visible
for(int index=0;index<1000;index++)
{
    //select random position and color for 640x480x8
    UCHAR color=rand()%256;
    Int x=rand()%640;
    Int y=rand()%480;

    //plot the pixel
    video_buffer[x+Y*mempitch]=color;
} // end for index

//now unlock the primary surface
lpddsprimary->Unlock(NULL);

```

当然，我忽略了所有的窗口初始化和事件循环，但那些都是一成不变的东西。完整的代码请参考一下 CD 上的 DEMO6.CPP 和相应的可执行文件 DEMO6.EXE。它们包含了前述注入控制台游戏 Game_Main()函数中的代码，同 Game_Init()一起显示在下面。图 6.12 是一个程序执行时的屏幕照片。

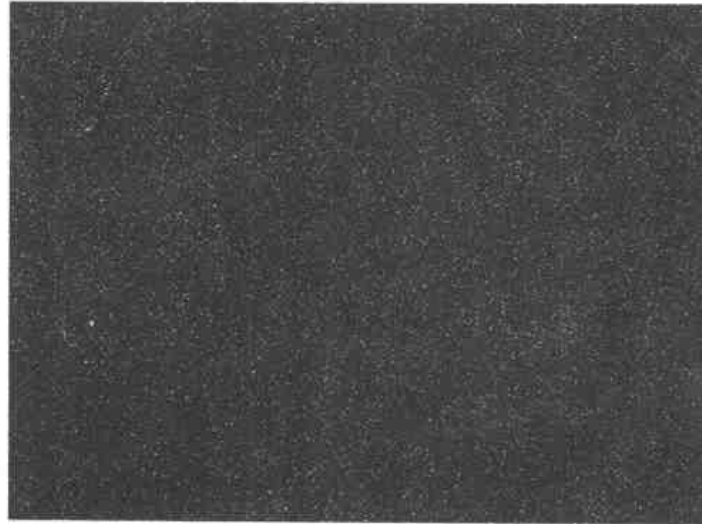


图 6.12 执行 DEMO6.EXE

```

Int Game_Main(void *parmas=NULL,int num_parms=0)
{
//this is the main loop of the game, do all your processing
//here

//for now test if user is hitting ESC and send WM_CLOSE
if (KEYDOWN(VK_ESCAPE))
    SendMessage(main_window_handle,WM_CLOSE, 0,0);

//plot 1000 random pixels to the primary surface and return
//clear ddsd and set size ,never assume it's clean
memset(&ddsd, 0,sizeof(ddsd));
ddsd.dwSize=sizeof(ddsd);

if (FAILED(lpddsprimary->Lock(NULL,&ddsd,
    DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT,
    NULL)))
{
    //error
    return (0);
} // end if

//now ddsd.lPitch is valid and so is dds.lpSurface

//make a couple aliases to make code cleaner,so we don't
//have to cast
int mempitch=(int)ddsd.lPitch;
UCHAR *video_buffer=(UCHAR *)ddsd.lpSurface;

//plot 1000 random pixels with random colors on the
//primary surface,they will be instantly visible

```

```

for(int index=0;index<1000;index++)
{
    //select random position and color for 640x480x8
    UCHAR color=rand()%256;
    Int x=rand()%640;
    Int y=rand()%480;

    //plot the pixel
    video_buffer[x+Y*mempitch]=color;
} // end for index

//now unlock the primary surface
if(FAILED(lpddsprimary->Unlock(NULL)))
return(0);

//Sleep a bit
Sleep(30);

//return success or failure or your own return code here
return(1);

} //end Game_Main
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

int Game_Init(void *parms=NULL,int num_parms=0)
{
    //this is called once after the initial window is created and
    //before the main event loop is entered ,do all your initialization
    //here

    //first create base IDirectDraw interface
    if (FAILED(DirectDrawCreate(NULL,&lpdd,NULL)))
    {
        //error
        return(0);
    } //end if
    //now query for IDirectDraw4
    if (FAILED(lpdd->QueryInterface(IID_IDirectDraw4,
                                   (LPVOID *)&lpdd4)))
    {
        //error
        return(0);
    } // end if

    //set cooperation to full screen
    if (FAILED(lpdd4->SetCooperativeLevel(main_window_handle,
                                         DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX |
                                         DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)))
    {

```

```

        //error
        return(0);
    >// end if
// set display mode to 640x480x8
if (FAILED(lpdd4->SetDisplayMode(SCREEN_WIDTH,
                                SCREEN_HEIGHT,SCREEN_BPP, 0, 0)))
    {
        //error
        return(0);
    }//end if

//clear ddsd and set size
memset(&ddsd, 0,sizeof(ddsd));
ddsd.dwFlags=DDSD_CAPS;

//request primary surface
ddsd.ddsCaps.dwCaps=DDSCAPS_PRIMARYSURFACE;

//create the primary surface
if (FAILED(lpdd4->CreateSurface(&ddsd,&lpddsprimary,NULL)))
    {
        //error
        return(0);
    }// end if

//build up the palette data array
for (int color=1;color<255;color++)
    {
        //fill with random RGB values
        palette[color].peRed=rand()%256;
        palette[color].peGreen=rand()%256;
        palette[color].peBlue=rand()%256;
//set flags field to PC_NOCOLLAPSE
palette[color].peFlags=PC_NOCOLLAPSE;
    }//end for color

//now fill in entry0and 255 with black and white
palette[0].peRed=0;
palette[0].peGreen=0;
palette[0].peBlue=0;
palette[0].peFlags=PC_NOCOLLAPSE;;
palette[255].peRed =255;
palette[255].peGreen =255;
palette[255].peBlue=255;
palette[255].peFlags=PC_NOCOLLAPSE;

//creat the palette object
if (FAILED(lpdd4->CreatePalette(DDPCAPS_8BIT | DDPCAPS_ALLOW256 |
                                DDPCAPS_INITIALIZE,

```



```

        Palette, &lpddpal, NULL)))
{
    //error
    return(0);
} //end if

//finally attach the palette to the primary surface
if (FAILED(lpddsprimary->SetPalette(lpddpal)))
{
    // error
    return(0);
} // end if

// return success or failure or your own return code here
return(0);

) // end Game_Init

```

需要注意演示程序中的创建主窗体部分:

```

// create the window
if (!(hwnd = CreateWindowEx(NULL,          //extended style
                            WINDOW_CLASS_NAME, //class
                            "T3D DirectX pixel Demo", //title
                            WS_POPUP | WS_VISIBLE,
                            0, 0,          // initial x,y
                            640,480,       // initial width , height
                            NULL,          // handle to parent
                            NULL,          // handle to menu
                            Hinstance,     // instance of this application
                            NULL)))        // extra creation parms
    Return(0);

```

注意演示程序是使用 **WS_POPUP** 窗口风格, 而不是 **WS_OVERLAPPEDWINDOW** 风格。你可能记得这种风格不带有任意控件和 **WindowsGUI** 痕迹, 这便是你所想要的全屏 **DirectX** 应用。

清 除

在结束这一章之前, 我想再引出一个主题, 就是我推迟阐述的资源管理。哦, 这一并不有趣的概念只表示在你使用完 **DirectDraw** 或 **DirectX** 之后确定要用 **Release()** 释放掉。例如, 如果你看一下 **DEMO6_3.CPP** 的源代码, 你将看到在 **Game_Shutdown()** 函数中多次调用 **Release()** 来释放所有的 **DirectDraw** 对象, 将其返还给系统和 **DirectDraw** 本身, 代码如下:

```
int Game_Shutdown(void *parms=NULL,int num_parms=0)
{
//this is called after the game is exited and the main event
//loop while is exited ,do all you cleanup and shutdown here
// first the palette
if (lpddpal)
{
    lpddpal->Release();
    lpddpal=NULL;
} //end if
// now the primary surface
if (lpddsprimary)
{
    lpddsprimary->Release();
    lpddsprimary=NULL;
} // end if
//now blow away the IDirectDraw4 interface
if (lpdd4)
{
    lpdd4->Release();
    lpdd4=NULL;
} //end if
//return success or failure or your own return code here
return(1);
} // end Game_Shutdown
```

通常，在使用完对象后就应该释放它们，并且释放的顺序要与创建的顺序相反。例如，你依次创建 **DirectDraw** 对象、主画面、调色板，那么释放顺序就应该是调色板、主画面、**DirectDraw** 对象，代码如下：

```
// first kill the palette
if (lpddpal)
{
    lpddpal->Release();
    lpddpal=NULL;
} // end if

// now the primary surface
if (lpddsprimary)
```

```
lpddsprimary->Release();

//and finally the directdraw object itself
if (lpdd4)
{
    lpdd4->Release();
    lpdd4=NULL;
} //end if
```

警告

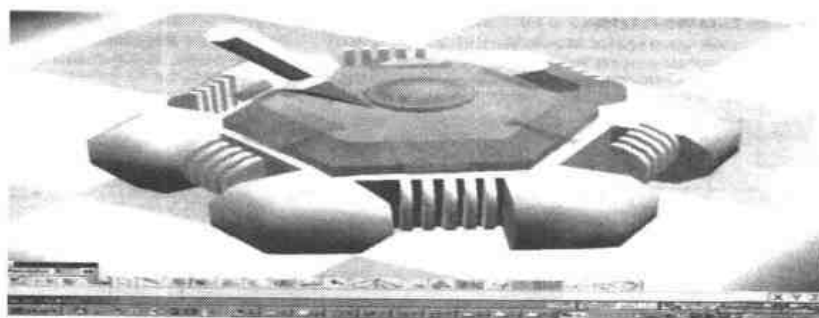


在调用Release()之前，注意界面是不是非空。这是完全有必要的，因为界面指针可以为空，如果执行程序时没有考虑到这一点而释放一个空值，将会产生错误。

总 结

本章讨论了 DirectDraw 的基本知识——如何启动及以全屏模式运行（主要部分）。当然，我们也简要讨论了调色板、显示画面及全屏和窗口化应用的区别。下一章，我们将会学到更多的东西。

7



高级 DirectDraw 和位图图形

本章将讲述 DirectDraw 的内容，并开始编写本书中所有演示程序和游戏的基础——图形库模块（T3DLIB1.CPP/H）。本章内容丰富，在本章课程中，我将给出一个图形库。尽管我们所要探讨的问题相当复杂，但我保证深入浅出地阐明这些问题。本章主要内容有：

- ➔ 真彩色模式
- ➔ 页翻转和双缓冲
- ➔ BLITTER
- ➔ 剪切
- ➔ 位图装载
- ➔ 彩色动画
- ➔ DirectX 窗口技术
- ➔ 从 DirectX 中获取信息

真彩色模式下工作

真彩色模式的色彩位数大于 8 位。它看起来当然要比 256 色精彩。但是，由于一些原因，它很少被用在 3D 引擎中。最大的问题是：

计算速度——标准的 640×480 像素缓冲有 307200 个像素。如果是 8 位，就意味着多数计算可通过单字节进行，且着色简单。而在 16 位或者 24 位模式下，全部 RGB 像素需要经过计算来获得（或者利用巨大的查询表）。但是，速度上至少慢上一倍。另外，每个像素点需要 2 到 3 个字节，不能够像 8 位那样一个字节。

当然，通过硬件加速，这对位图或者 3D（实际上多数 3D 卡采用 24/32 位色彩）来说并不是问题，但如果采用软件来着色（这将在本书中学到），就是一个大问题了。你想要对每个像素的操作最少，8 位像素最适合你（虽然 16 位可能更好看）。如果采用 8 位，可以确保具有像奔腾 70~100 的玩家也可以玩你的游戏，而无需担心玩家是否有奔腾 233MMX 或者 3D 加速设备。

内存带宽——这个问题是许多人没有考虑到的，你的 PC 机可能是 ISA、VLB、PCI 或者 PCI/AGP 等总线结构。除了 AGP 接口，其他连接方式相对于视频时钟来说要慢。这就意味着，如果你采用 PCI 总线，就会成为瓶颈。即使你有奔腾 III500，仍然于事无补。它会阻碍你使用 RAM 和加速硬件。当然，一些硬件的优化是有益的，如缓冲、多口 VRAM 等等。但不论你如何做，总会有一个界限在限制你。问题的本质是随着分辨率和色素深度的增加，很多时候带宽问题成为一个主要制约因素，而不是 CPU 的速度。但是，如果有 2 倍或者 4 倍 AGP 卡，就不成问题了。

现在，我已经给你讲述了适当采用真彩色方式的基本点，我将详细教给你如何来运用它们。我已决定采用 8 位模式使内容或 3D 软件更好理解（没有真彩色的 RGB 运算，3D 很难理解）。现在，就开始吧。

采用真彩色模式的概念同调色板模式类似，应注意的一点是你无需将真彩色的索引值写进屏幕缓冲，而是整个采用 RGB 像素的代码值。这意味着你必须知道真彩色模式的 RGB 编码。图 7.1 给出了 16 像素编码的一些例子。

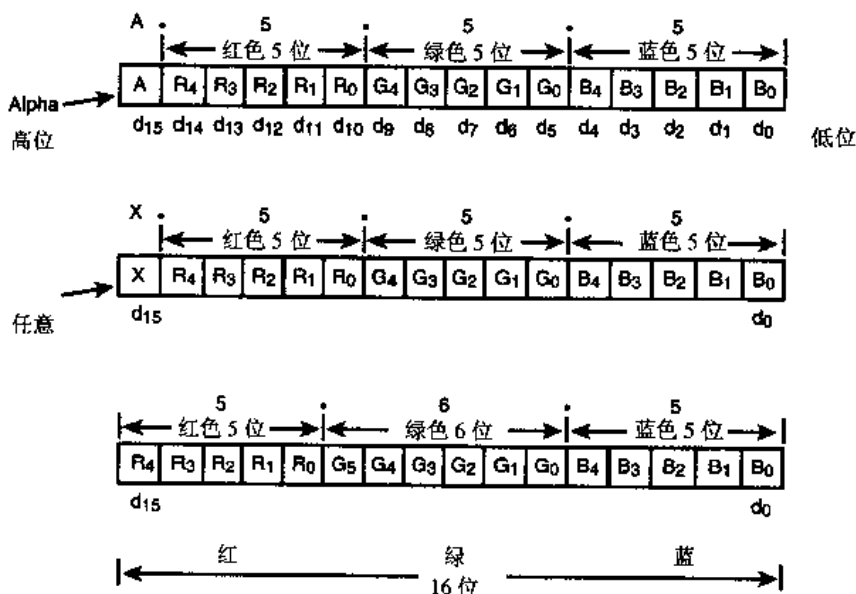


图 7.1 16 位像素的编码

16 位真彩色模式

如图 7.1 所示，16 位模式有许多可能的位编码技术。

Alpha.5.5.5——这种模式用 D_{15} 位存储一个 Alpha 值（表示是否透明），其余 15 位均匀分配给红色 5 位，绿色 5 位，蓝色 5 位。这样每种色彩产生的变换为 $2^5 = 32$ ，每个调色板有 $32 \times 32 \times 32 = 32768$ 种变换。

X.5.5.5——这种模式同 Alpha.5.5.5 类似，只是 MSB 位（最高位）没有用。依然是每种色彩有 32 种变化，共有 $32 \times 32 \times 32 = 32768$ 种色彩。

5.6.5——这是 16 位色彩最常见的模式。当然是 5 位分配给红色，6 位分配给绿色，5 位分配给蓝色。共有 $32 \times 64 \times 32 = 65536$ 种色彩。你可能会问，为什么给绿色 6 位？因为人的眼睛对绿色最为敏感，所以拓宽了绿色的位数。

现在知道了 RGB 色彩的编码方式，现在的问题是如何建立它们，你可以通过简单的移动或者屏蔽操作来完成这个任务，如下面的宏所示：

```
// this builds a 16 bit color value in 5.5.5 format(1-bit alpha mode)
#define _RGB16BIT555(r,g,b) ((b<32)+(g<32)<<5)+(r<32)<<10)

// this builds a 16 bit color value in 5.6.5 format(green dominate mode)
#define _RGB16BIT565(r,g,b) ((b<32)+(g<64)<<6)+(r<32)<<11)
```

从宏和图 7.2 知道，红色在高 5 位，绿色在中间 5 位，蓝色在最后 5 位。这对 PC 机来说是一种后退。因为它存放数据通常是从低位到高位排列，而这时却相反。但是这样更好，因为它使 RGB 的顺序同 MSB 到 LSB 的顺序相同。

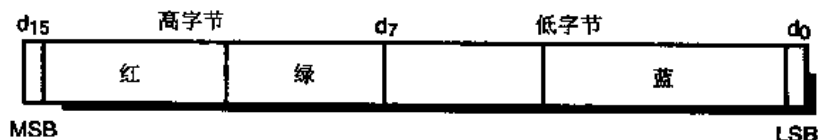


图 7.2 高位向低位排列色彩字

提示



在建立快速的 16 位演示程序之前，我必须强调一个小的细节。你如何检测系统到底是 5.5.5 或者 5.6.5 的显示模式呢？这很重要，它不受你的控制。你可以让 DirectDraw 工作在 16 位模式，但是具体的分配由硬件来决定。你又必须知道这个细节，因为绿色通道不小心会被阻塞。你需要知道的是像素点的格式。

获取像素格式

要知道画面的像素点格式，需要如下调用 `IDIRECTDRAWSURFACE4::GetPixelFormat()` 函数。

```
HRESULT GetPixelFormat(LPDDPIXELFORMAT lpDDPixelFormat);
```

你在上一章已经看到 `DDPIXELFORMAT` 的结构。但是你感兴趣的地方在于：

```

DWORD dwSize; // the size of the structure, must be set by you
DWORD dwFlags; // flags describing the surface, refer to Table 7.1
DWORD dwRGBBitCount; // number of bits for Red, Green, and Blue

```

在访问 DDPIXELFORMAT 的大小之前，必须设置 dwSize。调用后，dwFlags 和 dwRGBBitCount 才有效，包含有标志和 RGB 位数信息。表 7.1 给出了在 dwFlags 中可能包含的一些信息。

表 7.1 DDPIXELFORMAT.dwFlags 的有效标志

值	描 述
DDPF_ALPHA	像素格式描述一个单 - alpha 画面
DDPF_ALPHAPIXELS	画面有 alpha 信息的像素格式
DDPF_LUMINANCE	像素格式中有单一透明或者透明 alpha 分量的画面
DDPF_PALETTEINDEXED1	画面是 1 位色彩索引
DDPF_PALETTEINDEXED2	画面是 2 位色彩索引
DDPF_PALETTEINDEXED4	画面是 4 位色彩索引
DDPF_PALETTEINDEXED8	画面是 8 位色彩索引
DDPF_PALETTEINDEXEDTO8	画面是 1 位、2 位、4 位色彩索引到 8 位调色板
DDPF_RGB	像素格式中的 RGB 数据有效
DDPF_ZBUFFER	像素格式描述一个 Z 缓冲画面
DDPF_ZPIXELS	画面在像素中含有 Z 信息

注意：另外还有许多标志，尤其是与 D3D 相关的性能，更详细的信息请参考 DirectX SDK。

其中最重要的标志是：

DDPF_PALETTEINDEXED8——它标志着画面的色彩采用的是 8 位模式。

DDPF_RGB——它标志着画面色彩采用的是 RGB 模式，其格式可以通过 dwRGBBitCount 位查询。

所以，需要你做的工作就是像下面这样写一个测试程序。

```

LPDIRECTDRAWSURFACE4 lpdds_primary; // assume this is valid

// clear our structure
memset(&ddpixel, 0, sizeof(ddpixel));

// set length
ddpixel.dwSize = sizeof(ddpixel);

// make call off surface (assume primary this time)
lpdds_primary->GetPixelFormat(&ddpixel);

```

```
// now perform tests
// check if this is an RGB mode or palettized
if (ddpixel.dwFlags & DDPF_RGB)
{
    // RGB mode
    // what's the RGB mode
    switch(ddpixel.dwRGBBitCount)
    {
        case 15: // must be 5.5.5 mode
        {
            // use the _RGB16BIT555(r,g,b) macro
            } break;

        case 16: // must be 5.6.5 mode
        {
            // use the _RGB16BIT565(r,g,b) macro
            } break;

        case 24: // must be 8.8.8 mode
        {
            } break;

        case 32: // must be alpha(8).8.8.8 mode
        {
            } break;

        default: break;

    } // end switch

} // end if
else
if (ddpixel.dwFlags & DDPF_PALETTEINDEXED8)
{
    // 256 color palettized mode
} // end if
else
{
    // something else??? more tests
} // end else
```

代码相当简单，哈哈！有点难看，但是好用。当你以窗口模式创建主画面，并没有进行视频模式设置的时候，你就会发现 `GetPixelFormat()` 函数的真正威力。那时，你会对视频的性质无所适从，从而不得不查询系统。另外，你也不知道色彩深度、像素格式或者系统的分辨率。

现在你是一个 16 位的专家了。看下面的演示程序。除了采用 16 位色彩深度调用了 `SetDisplayMode()` 函数外，16 位应用程序其实也没有什么。作为一个例子，它会引导你利用

DirectDraw 建立一个 16 位全屏程序。

```
LPDIRECTDRAW lpdd_temp = NULL; // used to get directdraw1
LPDIRECTDRAW4 lpdd      = NULL; // used to get directdraw4
DDSURFACEDESC2 ddsd;      // surface description
LPDIRECTDRAW_SURFACE4 lpddsprimary = NULL; // primary surface

// create IDirectDrawdirectdraw interface 1.0 object and test for error
if (FAILED(DirectDrawCreate(NULL,&lpdd_temp,NULL)))
    return(0);

// now query for IDirectDraw4
if (FAILED(lpdd_temp->QueryInterface(IID_IDirectDraw4,
                                     (LPVOID *)&lpdd)))
    return(0);

// set cooperation level to requested mode
if (FAILED(lpdd->SetCooperativeLevel(main_window_handle,
                                     DDSCL_ALLOWMODEX | DDSCL_FULLSCREEN |
                                     DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)))
    return(0);

// set the display mode to 16 bit color mode
if (FAILED(lpdd->SetDisplayMode(640,480,16,0,0)))
    return(0);

// Create the primary surface
memset(&ddsd,0,sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSCL_CAPS;

// set caps for primary surface
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// create the primary surface
lpdd->CreateSurface(&ddsd,&lpddsprimary,NULL);
```

完成了。现在，你可以看到一个黑色的屏幕（如果主缓冲内存有数据的话，也可能有一些垃圾）。

为了简化讨论，假设你已经进行了像素格式的测试，并发现是 RGB16 位 5.6.5 模式，这也正确，因为你进行了格式设置。最坏的情节是你得到的是 5.5.5 格式。无论如何，向屏幕上写点需要：

1. 锁定画面，在本例中，它意味着你通过调用 Lock() 函数锁定了主画面。
2. 建立 16 位 RGB 字。这可以通过继承一个宏或者你自己来完成它。一般的，你给

像素绘制函数发送红色、绿色、蓝色值。必须对它们进行缩放，连接到主画面需要的 16 位 5.6.5 格式中。

3. 写像素点。这意味着采用一个 USHORT 指针定位主缓冲，写像素点到 VRAM 缓冲中。

4. 解锁主画面，调用 Unlock()。

下面给出 16 位像素绘制函数的代码。

```
void Plot_Pixel16(int x, int y, int red, int green, int blue,
                  LPDIRECTDRAWSURFACE4 lpdds)
{
    // this function plots a pixel in 16-bit color mode
    // very inefficient...

    DDSURFACEDESC2 ddsd; // directdraw surface description

    // first build up color WORD
    USHORT pixel = _RGB16BIT565(red,green,blue);

    // now lock video buffer
    DDRAW_INIT_STRUCT(ddsd);

    lpdds->Lock(NULL,&ddsd,DDLOCK_WAIT |
               DDLOCK_SURFACEMEMORYPTR,NULL);

    // write the pixel

    // alias the surface memory pointer to a USHORT ptr
    USHORT *video_buffer = ddsd.lpSurface;

    // write the data
    video_buffer[x + y*(ddsd.lPitch >> 1)] = pixel;

    // unlock the surface
    lpdds->Unlock(NULL);

} // end Plot_Pixel16
```

注意到用了 DDRAW_INIT_STRUCT(ddsd)，它是一个简单的宏，将结构初始为 0，设置 dwSize 标志位。下面是宏的定义。

```
// this macro should be on one line
#define DDRAW_INIT_STRUCT(ddstruct)
{ memset(&ddstruct, 0, sizeof(ddstruct));
  ddstruct.dwSize=sizeof(ddstruct); }
```

例如，用 RGB(255, 0, 0) 在主画面(10, 30)绘制绘制像素可以这样做：

```
Plot_Pixel16(10, 20, //x, y
255, 0, 0, // rgb
lpddsprimary); // surface to draw on
```

虽然看起来相当简单，但效率特别低。你可以利用一些优化措施进行优化。首要问题是每次函数都需要锁定或者解锁发送的画面。这不能被接受，锁定和解锁需要花费一些视频卡几百毫秒甚至更多。在一个游戏循环中，其基本原则是在进行所有操作之前对画面只锁定一次，然后完成操作后解锁一次。如图 7.3 所示。这样，你就不必一直进行锁定/解锁、内存清零等操作。例如，填充 DDSURFACEDESC2 结构花费的时间可能比画像素还要多。函数不是内联函数就更不用说了，前面的函数就足以毁了你的程序。

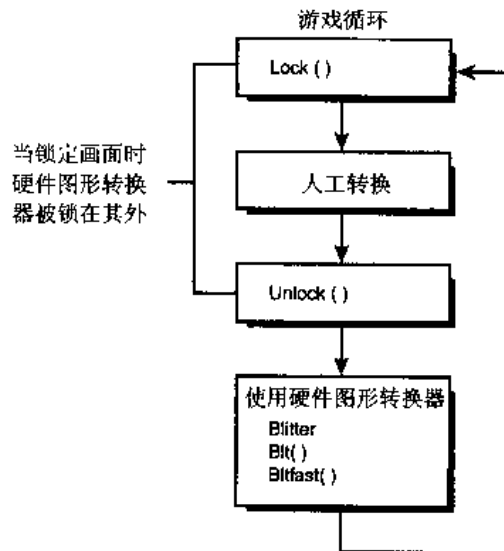


图 7.3 DirectDraw 应该被锁定得尽量小

这是游戏程序员应该时刻记住的一些事情。在此不是在编写字处理程序，因为你需要速度。下面是经过一点优化后的程序版本，但是它至少要快十倍。

```
inline void Plot_Pixel_Fast16(int x, int y,
int red, int green, int blue,
USHORT *video_buffer, int lpitch)
{
// this function plots a pixel in 16-bit color mode
// assuming that the caller already locked the surface
// and is sending a pointer and byte pitch to it
// first build up color WORD
USHORT pixel = _RGB16BIT565(red, green, blue);
// write the data
video_buffer[x+y*(lpitch>>1)] = pixel;
} // end Plot_Pixel_Fast16
```

我也不想进行乘法运算或者移位运算，但是这个新版本还不坏。你可以通过两种技巧去掉乘和移位。首先，因为 `lpitch` 是内存的字节宽度，移位是有必要的。但是，你现在的调用是假定已经锁定了画面，从画面查询内存指针和步长。没有脑子的人才会加上计算一个 **WORD** 或者 16 步长的语句，如下：

```
int lpitch16 = (lpitch>>1);
```

基本上，`lpitch16` 现在是一个组成视频的 16 位数。有新的值，就可以如下重新写一次函数。

```
inline void Plot_Pixel_Fast16(int x, int y,
int red, int green, int blue,
USHORT *video_buffer, int lpitch)
{
// this function plots a pixel in 16-bit color mode
// assuming that the caller already locked the surface
// and is sending a pointer and byte pitch to it
// first build up color WORD
USHORT pixel = _RGB16BIT565(red, green, blue);
// write the data
video_buffer[x+y*lpitch16] = pixel;
} // end Plot_Pixel_Fast16
```

就是这样，函数是内联函数，有一次乘法、加法和访问内存。不错，但是可以更好。最终的优化可以利用一个大的查询表代替乘法，但是，这可能没有必要，因为新一代的奔腾处理器乘法很简单。这也是提速的一种措施。

另外，我们可以通过一系列移位加法去掉乘法。例如，在线性内存模式下（每一线都没有跳越），你知道在 640×480 、16 位模式下，每一扫描线正好是 1280 个字节。因此，你需要把 y 乘以 640，因为排列会自动利用指针，将[]排列中的东西放大两倍，（每个 **USHORT WORD** 为两个字节。）数学上是这样的。

$$y \times 640 = y \times 512 + y \times 128$$

$512=2^9$ ， $128=2^7$ 。因此，如果将 Y 左移 9 次，将值同 Y 左移 7 次相加，结果就是 $Y \times 640$ 。数学表达：

$$\begin{aligned} y \times 640 &= y \times 512 + y \times 128 \\ &= (y \ll 9) + (y \ll 7) \end{aligned}$$

就是这样，如果你不熟悉这个技巧。可以参考图 7.4。一般的，将二进制数右移一位相当于除以 2，左移一位相当于乘以 2。但是，如果不是 2 的整数次幂，你可以像前面那样进行分解或整数次幂之和。

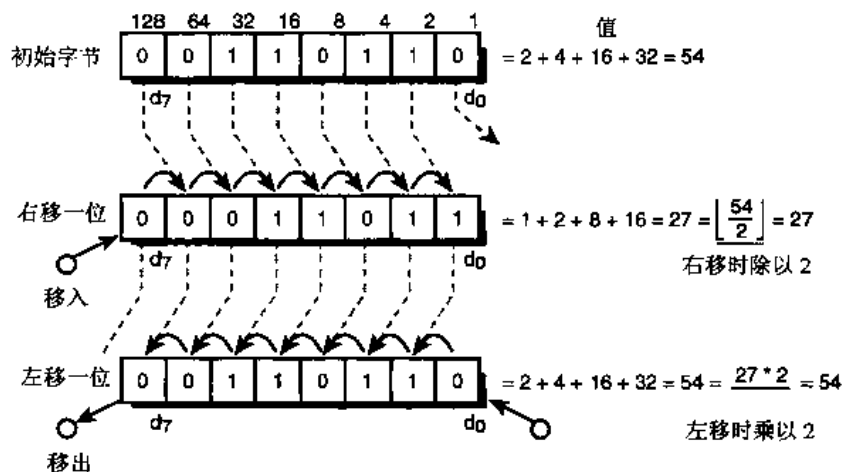


图 7.4 用位移动代替乘除运算

注意

在第十一章“算法、数据结构、内存管理及多线程”中，你会接触到更多的技巧。

用 16 位模式向屏幕写点的例子见 CD 中的 DEMO7_1.CPPIEXE。程序完成了你在下面描绘的一切，向屏幕随机画一些像素点。看看代码，你会发现你不需要调色板了，非常好。同时，代码是标准的 T3D 游戏引擎模板，需要你真正看的是 Game_Init()、Game_Main()。Game_Main()的代码如下：

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
    // this is the main loop of the game, do all your processing
    // here

    // for now test if user is hitting ESC and send WM_CLOSE
    if (KEYDOWN(VK_ESCAPE))
        SendMessage(main_window_handle, WM_CLOSE, 0, 0);

    // plot 1000 random pixels to the primary surface and return
    // clear ddsd and set size, never assume it's clean
    DDRAW_INIT_STRUCT(ddsd);

    // lock the primary surface
    if (FAILED(lpddsprimary->Lock(NULL, &ddsd,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT,
        NULL)))
        return(0);
}
```

```
// now ddsd.lPitch is valid and so is ddsd.lpSurface

// make a couple aliases to make code cleaner, so we don't
// have to cast
int lpitch16 = (int)(ddsd.lPitch >> 1);
USHORT *video_buffer = (USHORT *)ddsd.lpSurface;

// plot 1000 random pixels with random colors on the
// primary surface, they will be instantly visible
for (int index=0; index < 1000; index++)
{
    // select random position and color for 640x480x16
    int red   = rand()%256;
    int green = rand()%256;
    int blue  = rand()%256;
    int x     = rand()%640;
    int y     = rand()%480;

    // plot the pixel
    Plot_Pixel_Faster16(x,y,red,green,blue,video_buffer,lpitch16);

} // end for index

// now unlock the primary surface
if (FAILED(lpddsprimary->Unlock(NULL)))
    return(0);

// return success or failure or your own return code here
return(1);

} // end Game_Main
```

24/32 位真彩色模式

一旦你掌握了 16 位模式，24 位和 32 位也就不算什么了。我将从 24 位开始，因为它比 32 位要简单。每个 RGB 蓝色的 24 位模式正好是一个字节。256 种变换无一损失，总共可能的色彩为 $256 \times 256 \times 256 = 16.7M$ 。不必担心哪个通道用多少位数，红色、绿色、蓝色就像 16 位那样编码。

因为每通道一个字节，共三个通道，也就是每像素三个字节。这就使得定位困难。如图 7.5 所示。在 24 位模式下，写像素相当随意，如下所示：

```
inline void Plot_Pixel_Fast16(int x, int y,
int red, int green, int blue,
USHORT *video_buffer, int lpitch)
{
```

```

// this function plots a pixel in 24-bit color mode
// assuming that the caller already locked the surface
// and is sending a pointer and byte pitch to it

// in byte or 8-bit math the proper address is: 3*x+y*lpitch
// this is the address of the low order byte which is the Blue channel
// since the data is in RGB order
DWORD pixel_addr = (x+x+x)+y*lpitch;

// write the data, first blue
video_buffer[pixel_addr] = blue;

// write the data, first green
video_buffer[pixel_addr+1] = green;
// write the data, first red
video_buffer[pixel_addr+2] = red;
} // end Plot_Pixel_24

```

函数采用的参数有：X、Y、RGB 色彩、视频缓冲起始地址、字节为单位的内存步长。在发送内存步长或者以字长度计的视频缓冲时没有采用指针，原因是没有三字节长度的数据类型。因此函数基本上是从所需像素处开始定位视频缓冲，然后为像素写入红色、绿色、蓝色。

24 位的例子见 CD 中的 EDMO7-2.CPPIEXE。基本上以 24 为模式模仿了 DEMO7_1.CPPIEXE。

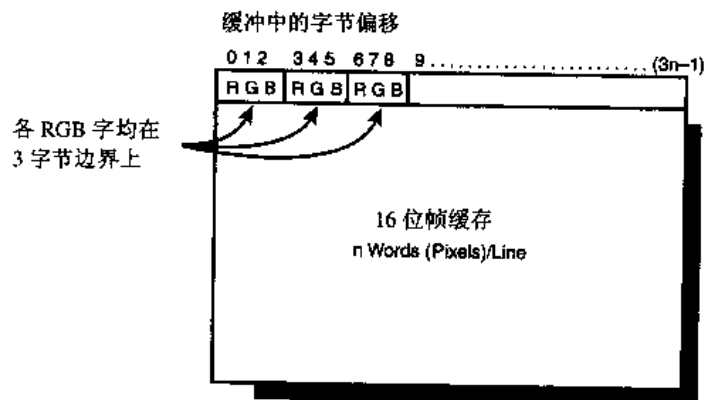


图 7.5 3 字节 RGB 寻址非常复杂

关于 32 位色彩，如图 7.6 所示，色彩设置略有不同。

Alpha(8).8.8.8——这种方式用 8 位来表示 Alpha 值或者透明信息（有时是一些其他信息）。然后是红色、绿色、蓝色，每种色彩 8 位。但是，简单的位图不用管这些，不考虑 Alpha 值，随便写进去 8 位就行。这种模式优点在于每像素 32 位，是奔腾处理器最快的可能内存寻址方式。

X(8).8.8.8—除了最高的 8 位被放弃之外，同上面的模式基本相同。但是，为了安全起见，我仍然建议你将它们赋值为 0。你会问，这种模式同 24 位相同，为什么还要它呢？答案是视频卡不能 3 位定位，四位只是为了对齐。

现在，看一下 32 位色彩字的宏定义：

```
// this builds a 32 bit color value in A.8.8.8 format (8-bit alpha mode)
#define _RGB32BIT(a, r, g, b) ((a)<<24)|((r)<<16)|((g)<<8)|b)
```

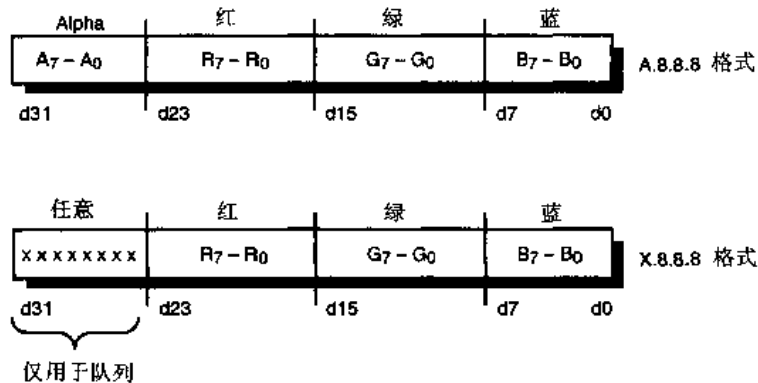


图 7.6 32 位色彩像素编码

需要你来做的就是你的像素绘制函数中使用新的宏，并采用每点 4 字节的数据。如下所示。

```
inline void Plot_Pixel_Fast16(int x, int y,
int red, int green, int blue,
USHORT *video_buffer, int lpitch)
{
// this function plots a pixel in 24-bit color mode
// assuming that the caller already locked the surface
// and is sending a pointer and byte pitch to it

// first build up color WORD
UINT pixel = _RGB32BIT(alpha, red, green, blue);

// write the data, first blue
video_buffer[x+y*lpitch32] = pixel;

} // end Plot_Pixel_32
```

它看起来应该很熟悉。惟一不同的是 `lpitch32` 是 4 的倍数，所以是 **DWORD** 或 32 位 **WORD** 步长，记住这点。看 **DEMO7_3.CPP\EXE**，它同样是像素绘制演示程序，不过是 32 位方式。它应该可以在你的计算机上运行，因为现在大多数显卡支持 32 位而不是 24 位模式。

好了，就不再就真彩色做过多的讨论，我想你已经能够用它们转变任何 8 位程序。记住，我不能够保证每个人都有奔腾 II450 和 VoodooII3D 加速卡。

采用 8 位色彩绘制的图像可以使更多的人能够使用。

双 缓 冲

你现在可以对主画面的内容做直接修改，通过视频控制，直接渲染每个屏幕画面。但是如何平滑地显示动画呢？这是一个明显的问题。让我来解释一下。就像我在本书中早些时候所说的那样，多数计算机动画通过在画面外的缓冲区域上画每个动画的备用屏幕，然后将图像快速地切换到显示画面实现。如图 7.7 所示。

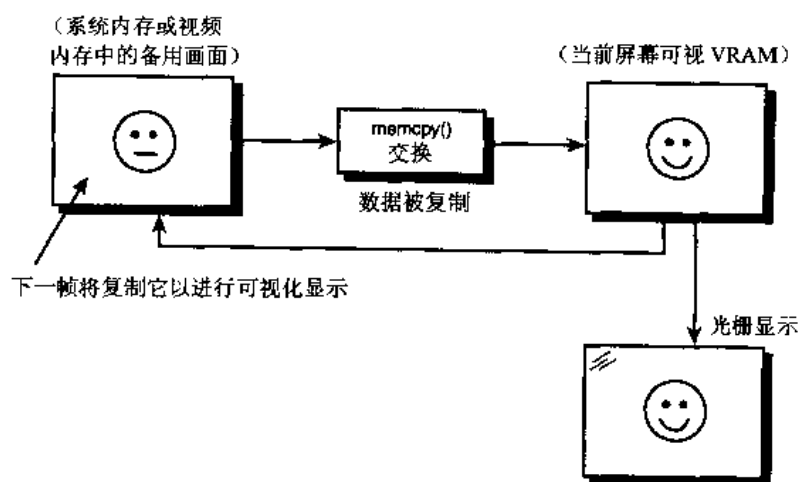


图 7.7 采用双缓冲技术运行动画

这种方法使用户看不到你擦画面、生面画面及其他一些对画面所做的事。只要是备用画面可以在足够短的时间内拷贝到显示画面，理论上你就可以在一秒钟内完成 15 次，或者称 15fps，并可以平滑运行游戏。但是，现在的标准至少是 30fps，所以现在我们来讨论一下高质量的动画技术。

在缓冲区域画出图像，然后拷贝到显示画面的技术被称为双缓冲技术，也是 99% 的游戏所采用的动画技术。但是，在过去（尤其是在 DOS 平台下），没有特殊的硬件帮助实现这个过程。随着 DirectX/DirectDraw 技术的介入，就有了明显的变化。

如果加速硬件存在（且在视频卡上有足够的 VRAM 内存），可以使用一种类似于双缓冲的页交换技术。页交换技术类似双缓冲技术，不过是在你画出一两个备用的可见画面之后，直接借助于硬件使其他画面激活可见。基本上去掉了拷贝的过程，因为用来指向光栅化的硬件寻址系统直接指向了不同的内存接口。结果是一个即时的页交换和可见屏幕的调整（这就是页交换的由来）。

当然，页交换总是可行的，许多程序员在进行 Mode X (320×240 , 320×400) 编程的时候就采用它。但是，它是一种低级、底层、直接的技术，通常需要汇编语言和视频控制编程来完成这项工作。但利用 DirectDraw 就可以轻而易举地完成，你将在下一部分接触到它。我只是想要你知道在我给出你双缓冲技术之前你处在什么位置。

实现双缓冲微不足道。需要你完成的只是制订一块同主 DirectDraw 画面相同尺寸的内存。将动画的每一屏幕画到它上面，然后拷贝双缓冲到主显示画面。遗憾的是，此方案有一个问题……

要创建一个 $640 \times 480 \times 8$ 的 DirectDraw 模式，就需要定位一个 640×480 的双缓冲或者说一个 307,200 字节的线性阵列。并记住数据以行顺序一行行向屏幕映像。下面就是建立双缓冲的代码。

```
UCHAR *double_buffer = (UCHAR )malloc(640*400);
```

或者用新的 C++ 语言操作符：

```
UCHAR *double_buffer = new UCHAR[640*400];
```

每种方法都可以得到一个双缓冲指针指向的 307200 字节大小的线性可寻址内存阵列。对一个位于 (x, y) 的像素点寻址，只需：

```
double_buffer[x+640*y] = _
```

看起来合情合理，因为每条实线有 640 字节，而且你用每线 640 字节共 480 线来保证矩形影像。好，下面有一个问题，假设你锁定一个指针 primary_buffer 指向主显示画面，并假设你在锁定内存过程中提取了内存步长，存储在 mempitch 中，如图 7.8 所示。如果 mempitch 等于 640，可以通过下面的代码拷贝 double_buffer 到 primary_buffer：

```
memcpy((void *)primary_buffer, (void *)double_buffer, 640*400);
```

同时，double_buffer 将立即在基本缓冲中出现。

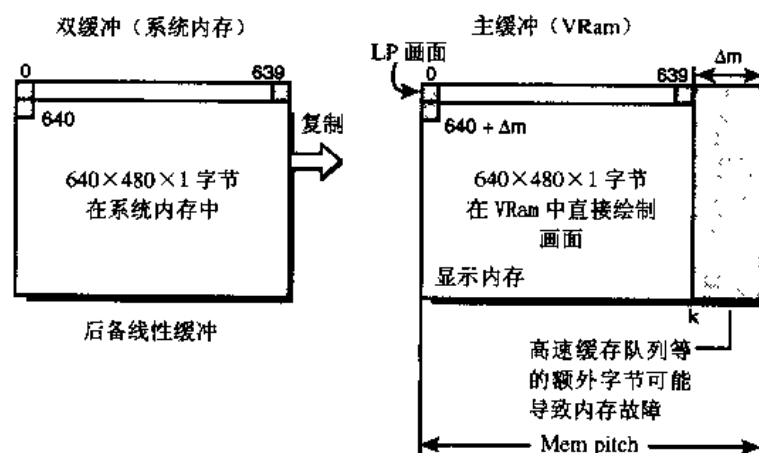


图 7.8 主画面可能每线有额外内存，发生寻址错误

技 巧



这里有一个潜在的优化措施，注意，我使用的是 `memcpy()`。此函数相当慢，因为它仅仅（在一些编译器上）拷贝字节。更好的方法是写一个你自己的 `DWORD` 或者 32 位拷贝程序。你可以采用 `inline` 或者额外的汇编语言。当到优化理论时，你就会明白怎样做。但是这是一个利用奔腾处理器能够以 32 位处理大块数据的一个例子。

并且，双缓冲立刻在主存储中出现。一切看来很好，错！`memcpy()` 代码只是在 `mempitch` 或者主画面的步长为刚好每线 640 字节时才工作。这可能对也可能不对。前面的 `memcpy()` 代码可能导致很严重的错误。一个更好的双缓冲拷贝函数是加一个小函数，检测主画面的步长是否 640。是，就采用 `memcpy()` 函数，不是，就一行行拷贝。虽然慢了一点，但却能够做到最好……下面是程序代码：

```
// can we use a straight memory copy?
if (mempitch == 640)
{
    memcpy ((void *)primary_buffer, (void *)double_buffer, 640*400);
} // end if
else
{
    // copy line by line, bummer!
    for (int y=0; y<480; y++)
    {
        // copy next line of 640 bytes
        memcpy ((void *)primary_buffer, (void *)double_buffer, 640*400);

        // now for the tricky part...
        // advance each pointer ahead to next line
        primary_buffer+=mempitch;

        // we know that we need to advance 640 bytes per line
        double_buffer+=640;
    } // end for y
} // end else
```

图 7.9 给出了进程的图例。如你所见，这是你不得不完成的多次工作中的一次。但是，你至少可以将代码优化成为 4 字节或 32 位拷贝代码。这还使我感到有点欣慰。

作为一个例子，我也创建了一个演示程序，在双缓冲中绘制了一些随机像素，然后以 $640 \times 480 \times 8$ 的模式拷贝到主缓冲。在拷贝中有一个很长时间的延时，使你能够看到两个画面的区别。程序的名字是 `DEMO7_4.CPP|EXE`，在 CD 上。记住，你自己编译时需要在工程中加上 `DDRAW.LIB`，并将 `DirectX` 头文件路径设对。下面是产生所有动作的程序中的 `Game_Main()`。

```
int Game_Main(void *parms = NULL, int num_parms = 0)
{
    // this is the main loop of the game, do all your processing
    // here
```

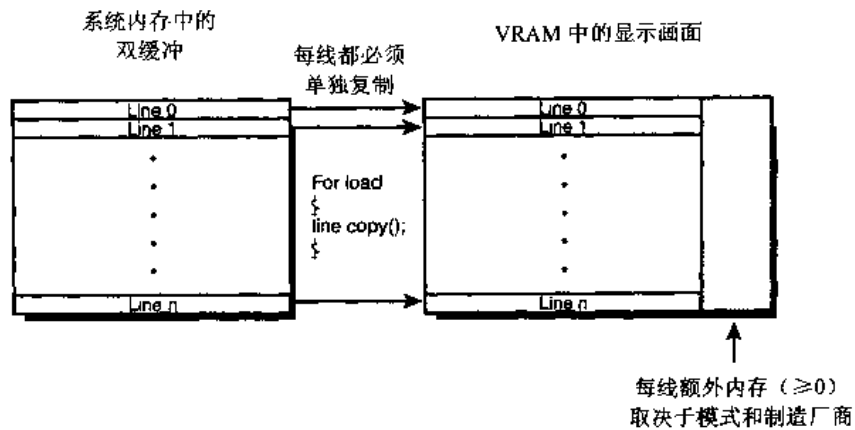


图 7.9 逐线复制双缓冲

```

UCHAR *primary_buffer = NULL; // used as alias to primary surface buffer

// make sure this isn't executed again
if (window_closed)
    return(0);

// for now test if user is hitting ESC and send WM_CLOSE
if (KEYDOWN(VK_ESCAPE))
{
    PostMessage(main_window_handle, WM_CLOSE, 0, 0);
    window_closed = 1;
} // end if

// erase double buffer
memset((void *)double_buffer, 0, SCREEN_WIDTH*SCREEN_HEIGHT);

// you would perform game logic...

// draw the next frame into the double buffer
// plot 5000 random pixels
for (int index=0; index < 5000; index++)
{
    int x = rand()%SCREEN_WIDTH;
    int y = rand()%SCREEN_HEIGHT;
    UCHAR col = rand()%256;
    double_buffer[x+y*SCREEN_WIDTH] = col;
} // end for index

// copy the double buffer into the primary buffer
DDRAW_INIT_STRUCT(ddsd);

// lock the primary surface
lpddsprimary->Lock(NULL, &ddsd,
    DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL);
    
```

```

// get video pointer to primary surface
primary_buffer = (UCHAR *)ddsd.lpSurface;

// test if memory is linear
if (ddsd.lPitch == SCREEN_WIDTH)
{
    // copy memory from double buffer to primary buffer
    memcpy((void *)primary_buffer, (void *)double_buffer,
        SCREEN_WIDTH*SCREEN_HEIGHT);
} // end if
else
{ // non-linear

    // make copy of source and destination addresses
    UCHAR *dest_ptr = primary_buffer;
    UCHAR *src_ptr = double_buffer;

    // memory is non-linear, copy line by line
    for (int y=0; y < SCREEN_HEIGHT; y++)
    {
        // copy line
        memcpy((void *)dest_ptr, (void *)src_ptr, SCREEN_WIDTH);

        // advance pointers to next line
        dest_ptr+=ddsd.lPitch;
        src_ptr +=SCREEN_WIDTH;

        // note: the above code can be replaced with the simpler
        // memcpy(&primary_buffer[y*ddsd.lPitch],
        //         double_buffer[y*SCREEN_WIDTH], SCREEN_WIDTH);
        // but it is much slower due to the recalculation
        // and multiplication each cycle

    } // end for

} // end else

// now unlock the primary surface
if (FAILED(lpddsprimary->Unlock(NULL)))
    return(0);

// wait a sec
Sleep(500);

// return success or failure or your own return code here
return(1);

} // end Game_Main

```

动态画面

到现在为止，我讲的都是创建不同类型的画面，但直到现在，你看到的还是主画面如何绘制。现在我想谈谈备用画面。基本上，有两种类型备用画面。第一种是后备缓冲（back buffer）。

后备缓冲是指几何形状、色彩深度同主画面相同的用在动画链中的画面。后备缓冲画面较为独特，因为当你创建主画面时就创建它们。它们是主画面页交换链中的一个。换句话说，当你需要一个或者多个备用画面时作为后备缓冲时，默认状态下，DirectDraw 会假定你利用它们做动画循环。图 7.10 给出了主画面和备用画面及后备缓冲的关系。

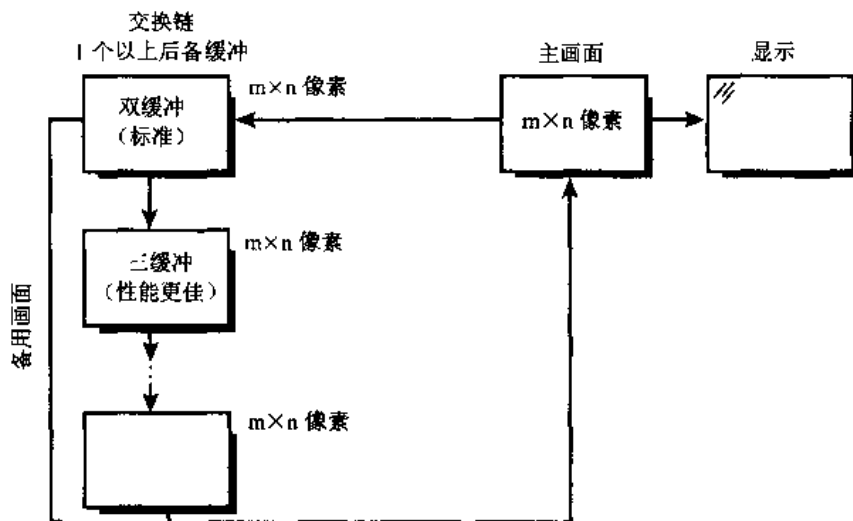


图 7.10 主画面和后备缓冲

你创建后备缓冲的目的是仿效双缓冲，但是这种方式更接近 DirectDraw。如果你创建了后备缓冲，它一般在 VRAM 中，非常快。另外，你可以同主画面一起对它实行页交换。比双缓冲方案下的内存拷贝快得多。

技术上，你可以具有你在交换链中想要的多个后备缓冲。但是，有时用完了 VRAM 就会在系统内存中创建画面，这会很慢。通常，如果你创建了一个 $m \times n$ 的具有一个字节的色彩深度的画面，主画面需要的内存量当然是 $m \times n$ 字节（除非需要内存步长对齐）。因此，如果你有一个额外的后备缓冲备用画面，你就要乘 2，因为后备缓冲同主缓冲一样大。所以就需要 $2 \times m \times n$ 个字节。最终，如果色彩深度是 16 位，你还得将计算放大两倍，同样，32 位要放大 4 倍。例如， $640 \times 480 \times 16$ 模式主缓冲需要：

宽 \times 高 \times 每像素字节数

$640 \times 480 \times 2 = 614400$ 字节

如果想要一个后备缓冲，就要将结果乘以 2，最终的字节数为：

$614400 \times 2 = 1228800$ 字节

大约是 1.2MB。因此，如果你只有 1MB 的卡，就不要用有 $640 \times 480 \times 16$ 位色彩模式的 VRAM 后备缓冲。现在多数卡至少有 2MB，所以我们用起来比较安全，但是测测卡上到底有多少可用内存比较好。这点可以通过调用 GetCaps 类函数来实现。本章最后再讲述这点。

为了创建一个带有后备缓冲的主画面，你不得不创建 DirectDraw 所称的复杂画面。下面是创建步骤。

1. 首先，你要将 DDSD_BACKBUFFERCOUNT 加到 dwFlags 标志字段，向 DirectDraw 表明 DDSURFACEDESC2 的 dwBackBufferCount 字段有效，以及含有后备缓冲的数目（本例中为 1）。
2. 其次，将控制标志 DDSCAPS_COMPLEX 和 DDSCAPS_FLIP 加到 ddsCaps.dwCaps 字段中的字 DDSURFACEDESC2 结构上。
3. 最后，像通常一样创建主画面。从它调用 IDirectDrawSurface4::GetAttachedSurface()，获得后备缓冲。如下所示：

```
HRESULT GetAttachedSurface( LPDDSCAPS2 lpDDSCaps,
    LPDIRECTDRAWSURFACE4 FAR *lpDDAttachedSurface );
```

lpDDSCaps 是 DDSCAPS2 结构中包含所需画面信息的标志，在这种情况下，你需要一个后备缓冲，应该这样设置：

```
DDSCAPS2 ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
```

或者直接用 DDSURFACEDESC2 结构中 DDSCAPS2 字段存放另一个变量，像下面这样：

```
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
```

下面的代码创建了一个带有一个后备缓冲交换链的主画面：

```
// assume we already have the directdraw object etc...

DDSURFACEDESC2 ddsd; // directdraw surface description
LPDIRECTDRAWSURFACE4 lpddsprimary = NULL; // primary surface
LPDIRECTDRAWSURFACE4 lpddsback = NULL; // back buffer

// clear ddsd and set size
DDRAW_INIT_STRUCT(ddsd);

// enable valid fields
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
```

```

// set the backbuffer count field to 1
ddsd.dwBackBufferCount = 1;

// request a complex, flippable
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                    DDSCAPS_COMPLEX | DDSCAPS_FLIP;

// create the primary surface
if (FAILED(lpdd4->CreateSurface(&ddsd, &lpddsprimary, NULL)))
    return(0);

// now query for attached surface from the primary surface

// this line is needed by the call
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;

if (FAILED(lpddsprimary->GetAttachedSurface(&ddsd.ddsCaps, &lpddsback);

```

这时，`lpddsprimary` 指向主画面，即现在可见的画面。`lpddsback` 指向后备缓冲画面，看不见。如图 7.11 所示。要利用后备缓冲，你可以同对主画面一样对它进行解锁/锁定。

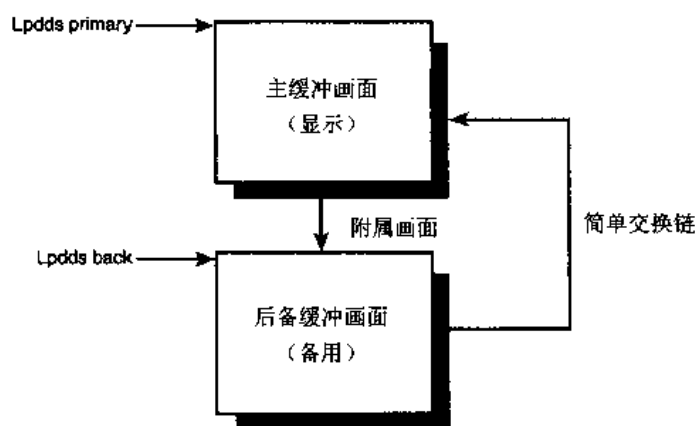


图 7.11 一个真正复杂的画面

所以，如果你想操纵后备缓冲中的信息，你可以这样做：

```

// copy the double buffer into the primary buffer
DDRAW_INPUT_STRUCT(ddsd);
// lock the back buffer surface
lpddsback->Unlock(NULL, &ddsd, DDLOCK_SURFACEMEMORYPTR|DDLOCK_WAIT, NULL);
// now ddsd.lpSurface and ddsd.lPitch are valid
// do whatever...
// unlock the back buffer, so hardware can work with it
lpddsback->Unlock(NULL);

```


现在，惟一的一个问题是你还不知道如何翻页。或者，将后备缓冲画面变成主画面从而产生两页动画。让我来教你。

页面变换

一旦创建了具有一个主画面和后备缓冲画面的复杂画面，就可以翻页了。标准动画循环需要以下几步（如图 7.12 所示）：

1. 清除后备缓冲。
2. 对后备缓冲屏幕渲染。
3. 用后备缓冲画面交换主画面。
4. 锁定刷屏速度（如 30fps）。
5. 回到第一步。

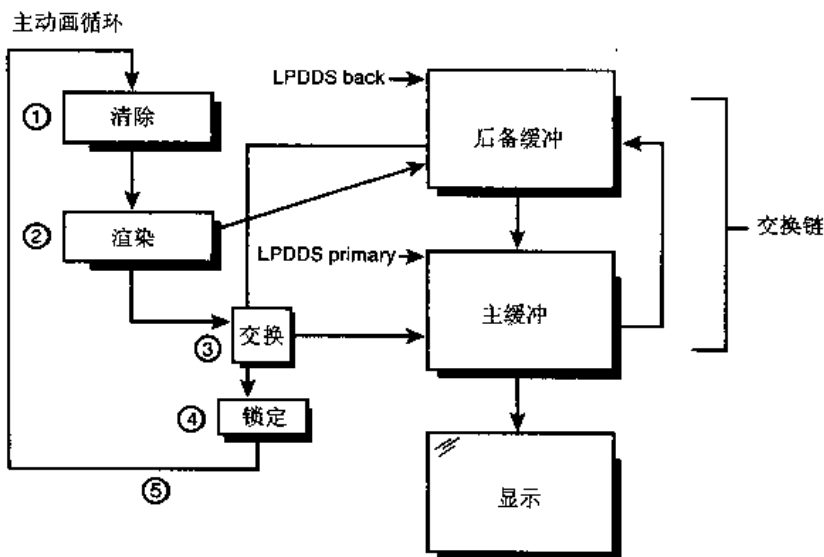


图 7.12 一个页交换动画系统

其中有一些使人迷惑的细节。首先，如果后备缓冲被交换到了主缓冲，后备缓冲会不会变成主画面，或者相反？如果这样，需要不需要在主画面绘制其他的每一帧？尽管这个问题很看似凶险，但实际上却不会发生。事实上，指向 VRAM 的指针是由硬件切换的，而且从 DirectDraw 和编程人员的观点来看，后备缓冲画面总是在屏幕以外，而主画面总是显示在屏幕上。所以你总是将每一帧画面绘制到后备缓冲中，并与主画面交换每一帧。

在交换链中，用下一个辅助画面交换主画面可以使用函数 `IDIRECTDRAW::Flip()`，如下所示：

```

HRESULT Flip(LPDDIRECTDRAW SURFACE4 lpDDSTargetOverride, // override surface
            DWORD dwFlags); // control flags
    
```

若操作成功则返回 DD_OK，否则返回错误代码。

参数非常简单，lpDDSurfaceTargetOverride 基本上是一个用来覆盖交换链的高级参数，交换到另外一个画面，而不是同主画面相连接的后背缓冲中的辅助画面进行交换，这里将其值设为 NULL。标志参数更应引起你的注意。表 7.2 给出了它的不同设置。

表 7.2 Flip() 函数的控制标志

值	描 述
DDFLIP_INTERVAL2	2 个垂直逆程后翻动
DDFLIP_INTERVAL3	3 个垂直逆程后翻动
DDFLIP_INTERVAL4	4 个垂直逆程后翻动

注意：默认情况是 1 个垂直逆程。

这些标志表明在两个交换页之间等待多少垂直逆程。默认值是 1。指定的垂直逆程数目达到了，DirectDraw 才为交换的每一页返回 DERR_WASSTILLDRAWING。如果设置 DDFLIP_INTERVAL2，DirectDraw 将每两个垂直同步后交换页面。如果 DDFLIP_INTERVAL3，将每三个垂直同步后换页。DDFLIP_INTERVAL4。将每四个垂直翻动后换页。

这些标志只是在设在 DDCAPS 结构中的 DDCAPS2_FLIPINTERVAL 设置返回设备后才起作用。

DDFLIP_NOVSYNC——此标志使得 DirectDraw 执行物理交换时尽量靠近下一个扫描线。

技 巧



能够创建一个复杂表面，具有两个后备缓冲，或一个共有包括主表面的三个表面的翻动链。这就称为三缓冲。它给出了性能的最终表现。问题很明显，如果你只有一个后备缓冲，显示硬件在利用它时可能会遇到你和硬件同时使用它的瓶颈。但是，翻动时另外有两个表面，硬件永远也不用等待。DirectDraw 三缓存的好处是你只需简单调用 Flip()，硬件以循环的方式翻动页表面，同时你还可以对一个后备缓冲着色，它对你来说是透明的。

DDFLIP_WAIT——此标志在有问题时，强迫硬件等待，直到可能有一个交换为止（而不是一有问题就立即返回）。

通常，你要为 DDFLIP_WAIT 设置标志。你必须像线程一样从主画面调用 Flip()，而不是在后备缓冲。这很合理，因为主画面是后备缓冲画面的“父画面”，后备缓冲是“父”交换链的一部分。下面给出了如何交换页的例子。

```
lpddsprimary->Flip(NULL, DDFLIP_WAIT);
```

并且，我发现，当程序运行出现一些愚蠢的错误时，加一个逻辑判断很有帮助。

```
while (FAILED(lpddsprimary->Flip(NULL, DDFLIP_WAIT)));
```

警告



在翻动时,主表面或者背后表面都必须解锁,所以在调用 Flip()翻动它们之前要确保对它们解锁了。

看 DEMO7_5.CPP\EXE 获得页交换的例子。我利用 DEMO7_4.CPP, 将双缓冲变成页交换, 当然, 我还调整了 Game_Init()代码创建了一个带有一个后备缓冲的复杂画面。下面是 Game_Init()、Game_Main()程序清单:

```
int Game_Init(void *parms = NULL, int num_parms = 0)
{
    // this is called once after the initial window is created and
    // before the main event loop is entered, do all your initialization
    // here

    LPDIRECTDRAW lpdd_temp;

    // first create base IDirectDraw interface
    if (FAILED(DirectDrawCreate(NULL, &lpdd_temp, NULL)))
        return(0);

    // now query for IDirectDraw4
    if (FAILED(lpdd_temp->QueryInterface(IID_IDirectDraw4,
        (LPVOID *)&lpdd4)))
        return(0);

    // set cooperation to full screen
    if (FAILED(lpdd4->SetCooperativeLevel(main_window_handle,
        DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX |
        DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)))
        return(0);

    // set display mode to 640X480X8
    if (FAILED(lpdd4->SetDisplayMode(SCREEN_WIDTH, SCREEN_HEIGHT,
        SCREEN_BPP, 0, 0)))
        return(0);

    // clear ddsd and set size
    DDRAW_INIT_STRUCT(ddsd);

    // enable valid fields
    ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

    // set the backbuffer count field to 1, use 2 for triple buffering
    ddsd.dwBackBufferCount = 1;

    // request a complex, flippable
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
```

```

        DDSCAPS_COMPLEX | DDSCAPS_FLIP;

// create the primary surface
if (FAILED(lpdd4->CreateSurface(&ddsd, &lpddsprimary, NULL)))
    return(0);

// now query for attached surface from the primary surface

// this line is needed by the call
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;

// get the attached back buffer surface
if (FAILED(lpddsprimary->GetAttachedSurface(&ddsd.ddsCaps, &lpddsback)));

// build up the palette data array
for (int color=1; color < 255; color++)
{
    // fill with random RGB values
    palette[color].peRed   = rand()%256;
    palette[color].peGreen = rand()%256;
    palette[color].peBlue  = rand()%256;

    // set flags field to PC_NOCOLLAPSE
    palette[color].peFlags = PC_NOCOLLAPSE;
} // end for color

// now fill in entry 0 and 255 with black and white
palette[0].peRed   = 0;
palette[0].peGreen = 0;
palette[0].peBlue  = 0;
palette[0].peFlags = PC_NOCOLLAPSE;

palette[255].peRed   = 255;
palette[255].peGreen = 255;
palette[255].peBlue  = 255;
palette[255].peFlags = PC_NOCOLLAPSE;

// create the palette object
if (FAILED(lpdd4->CreatePalette(DDPCAPS_8BIT | DDPCAPS_ALLOW256 |
                                DDPCAPS_INITIALIZE,
                                palette,&lpddpal, NULL)))
    return(0);

// finally attach the palette to the primary surface
if (FAILED(lpddsprimary->SetPalette(lpddpal)))
    return(0);

// return success or failure or your own return code here
return(1);

```

```

} // end Game_Init

////////////////////////////////////

int Game_Main(void *parms = NULL, int num_parms = 0)
{
    // this is the main loop of the game, do all your processing
    // here

    // make sure this isn't executed again
    if (window_closed)
        return(0);

    // for now test if user is hitting ESC and send WM_CLOSE
    if (KEYDOWN(VK_ESCAPE))
    {
        PostMessage(main_window_handle, WM_CLOSE, 0, 0);
        window_closed = 1;
    } // end if

    // lock the back buffer
    DDRAW_INIT_STRUCT(ddsd);
    lpddsback->Lock(NULL, &ddsd, DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL);

    // alias pointer to back buffer surface
    UCHAR *back_buffer = (UCHAR *)ddsd.lpSurface;

    // now clear the back buffer out

    // linear memory?
    if (ddsd.lPitch == SCREEN_WIDTH)
        memset(back_buffer, 0, SCREEN_WIDTH*SCREEN_HEIGHT);
    else
    {
        // non-linear memory

        // make copy of video pointer
        UCHAR *dest_ptr = back_buffer;

        // clear out memory one line at a time
        for (int y=0; y<SCREEN_HEIGHT; y++)
        {
            // clear next line
            memset(dest_ptr, 0, SCREEN_WIDTH);

            // advance pointer to next line
            dest_ptr += ddsd.lPitch;
        }
    }
}

```

```

        } // end for y

    } // end else

    // you would perform game logic...

    // draw the next frame into the back buffer, notice that we
    // must use the lpitch since it's a surface and may not be linear

    // plot 5000 random pixels
    for (int index=0; index < 5000; index++)
    {
        int x = rand()%SCREEN_WIDTH;
        int y = rand()%SCREEN_HEIGHT;
        UCHAR col = rand()%256;
        back_buffer[x+y*ddsd.lPitch] = col;
    } // end for index

    // unlock the back buffer
    if (FAILED(lpddsback->Unlock(NULL)))
        return(0);

    // perform the flip
    while (FAILED(lpddsprimary->Flip(NULL, DDFLIP_WAIT)));

    // wait a sec
    Sleep(500);

    // return success or failure or your own return code here
    return(1);

} // end Game_Main

```

Also, note the boldfaced code from Game_Main() that deals with the lock window_closed, reprinted here:

```

// make sure this isn't executed again
if (window_closed)
    return(0);

// for now test if user is hitting ESC and send WM_CLOSE
if (KEYDOWN(VK_ESCAPE))
{
    PostMessage(main_window_handle, WM_CLOSE, 0, 0);
    window_closed = 1;
} // end if

```

技巧



我在进程中加了 exit 语句, 以便即使在窗口销毁时, 也能使 Game_Main() 完成一次调用。这当然可能导致错误, 因为 DirectDraw 同窗口连接。因此, 我创建了一个锁定变量, 使得一旦窗口关闭, 大门仍然向 Game_Main() 函数打开。这是我应该在最后一个程序中就应该提及的重要细节, 但是我没有。当然, 我也可以另写一段, 但是我想给你演示一下 DirectX/Win32 同步编成是多么容易出错。

这就是关于页交换的全部。DirectDraw 完成了大部分工作, 但是我想给你留下最后一点细节。首先, 当你创建一个后备缓冲时, DirectDraw 可能在系统内存中创建它, 而不是在 VRAM 中 (如果没有剩余)。那时, 你无需做任何事情。DirectDraw 将会用双缓冲仿效页交换, 当调用 Flip() 时拷贝后备缓冲画面到主画面。但是, 会很慢。好处就是无论怎样, 你的代码都会运行。

注意



一般情况下, 你希望创建的主画面和背后缓存都在 VRAM 中进行。主画面总是在 VRAM 中, 但是可能粘在系统内存中的背后缓存上。但是一定要记住, VRAM 只有那么多。你可能在想提高图像速度时, 忘记 VRAM 缓存的在存放你的图像时的交换使用。用硬件从 VRAM 到 VRAM 中移动位图要比在系统内存中快得多。有时你有很多小的精灵或位图, 你可能决定采用系统内存做后备缓冲。那时, 你就遇到了麻烦, 双缓冲方案同 VRAM 动画技术中的页翻动不同, 它损失的时间, 远远超过了完全利用 VRAM 运行游戏所节省的时间。

应用图形变换器

假设你在 DOS 下编程, 如果没有从制造商那里或者笨重的第三方库函数获得显卡的驱动程序, 就不但不能利用 2D/3D 的硬件加速功能, 也会被局限于准 32 位世界中 (即使在 DOS 扩展版中)。从 DOOM 之后, 到处是硬件加速, 但是, 程序员可能很少应用它, 因为它是 Windows 的事情。但是, 利用 DirectX, 你可以利用所有的加速功能, 如图像、声音、输入、网络等等。但是, 最酷的是可以利用硬件图形变换进行位图移动或填充操作。例如图 7.13, 描述了一个 8×8, 256 色位图。如果你想将这个图像拷贝到视频或者备用缓冲的 (x, y) 处, 采用线性步长, 640×480 的分辨率。下面就是代码。

```
UCHAR *video_buffer; // points to VRAM of offscreen surface
UCHAR bitmap[8*8]; // holdsour bitmap in row major form
// crude bitmap copy
// outer loop is for each row
for (int index_y=0; index_y<8; index_y++)
{
    // inner loop for each pixel of each row
    for (int index_x=0; index_x<8; index_x++)
    {
        // copy the pixel without transparency
        video_buffer[x+index_x +(y+index_y)*640] =
        bitmap[index_x+index_y*8];
    } // end for index_x
```

```
} // end for index_y
```

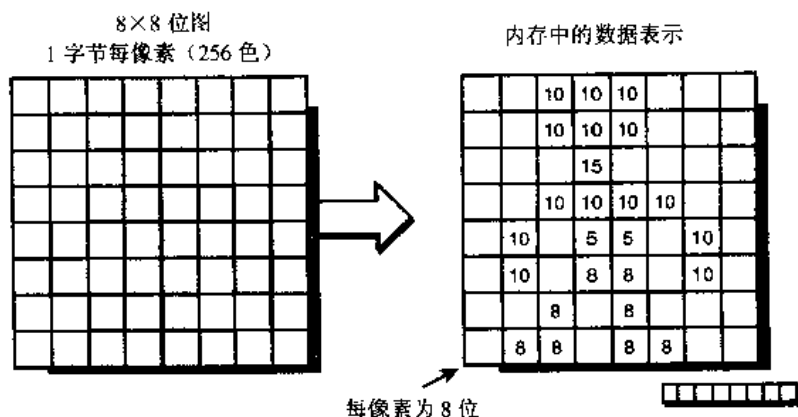


图 7.13 一个 8×8, 256 色位图

现在停顿一下, 确定你完全理解了它, 并且不看你也能够自己编写出来。看图 7.13 来帮助形象化。基本上你是将为矩形的像素位图从内存中一个地方拷贝到另外一个地方。此函数当然可以进行优化, 也存在一些问题。首先, 我想谈谈存在的问题。

问题 1: 函数运行太慢。

问题 2: 函数没有考虑到透明度, 也就是说如果你游戏中的物体处在黑色的环境中, 黑色也将一并被拷贝。问题如图 7.14 所示。你需要给它加上代码。

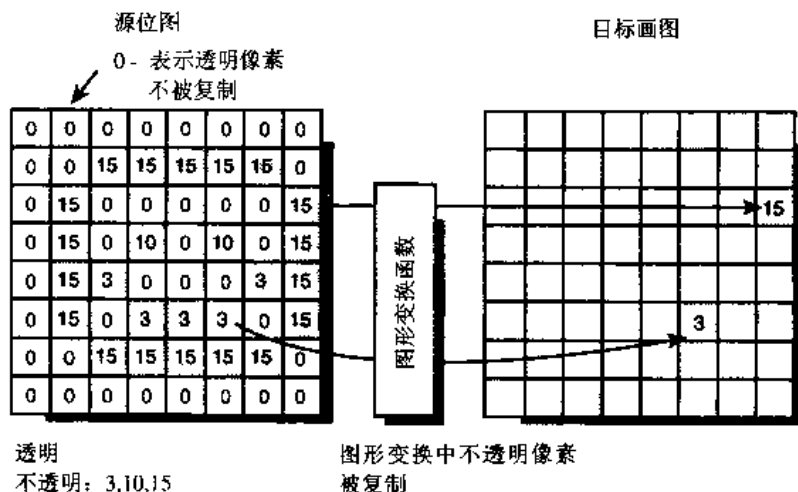


图 7.14 图形变换中不将透明点拷贝到主画面

在进行进一步优化之前, 你可以做下面的事情。

优化 1: 通过预先对源和目标内存的寻址, 去掉所有的乘法以及大多数加法, 然后对每个像素增加指针值。

优化 2: 利用内存填充非透明像素。

下面编写一个考虑了透明度的实际函数（用色彩 0）。采用更好的寻址方式，去掉乘法运算。这是例子：

```
void Blit8x8(int x, int y,
            UCHAR *video_buffer,
            UCHAR *bitmap)
{
    // this function blits the image sent in bitmap to the
    // destination surface pointed to by video_buffer
    // the function assumes a 640x480x8 mode with linear pitch

    // compute starting point into video buffer
    // video_buffer = video_buffer + (x + y*640)
    video_buffer += (x + (y << 9) + (y << 7));

    UCHAR pixel; // used to read/write pixels

    // main loop
    for (int index_y=0; index_y < 8; index_y++)
    {
        // inner loop, this is where it counts!
        for (int index_x=0; index_x < 8; index_x++)
        {
            // copy pixel, test for transparent though
            if ((pixel = bitmap[index_x])
                video_buffer[index_x] = pixel;
        } // end for index_x

        // advance pointers
        bitmap += 8; // next line in bitmap
        video_buffer += 640; // next line in video_buffer
    } // end for index_y
} // end Blit8x8
```

这个版本的图形变换函数很多情况下比前一个带有乘法运算得要快，并且这个程序处理的是带有透明像素的图像。练习的目的是给你演示这样一个事实：简单的事情可能会占用很多处理器周期。如果将循环计算在内，函数还有浪费。这就是前面的循环机制，当然程序还有点难看。必须进行透明测试，需要两个数组、一次写内存等等。这就是使用加速的原因。硬件图形变换可以在休息时完成此事，这也是你需要硬件图形变换图像的原因。这样你可以节省处理器周期做其他的事。

不用说，刚才的图形变换函数实际很蠢。对 $640 \times 480 \times 256$ 很难编码，没有进行任何剪切，只对 8 位位图有效。

现在，我已经给出了你过去画图的老方法。下面首次给出了图形变换，以及它是如何

对内存填充的。然后，你将会看到如何从一个画面向另外一个画面拷贝位图。本章最后，将使用图形变换技术画游戏中的物体，但要等一下。

使用图形变换器进行内存填充

虽然在 DirectDraw 中利用图形变换技术同人工编程来比微不足道，它依然是一个相当复杂的硬件。我每当着手讲解新的视频硬件时，在给你全貌之前都喜欢从简单的开始。因此让我们来做些十分有用的事情——内存填充。

内存填充意味着用一定的值填充 VRAM 中的一块区域。你已经在锁定画面，利用 memset() 或 memcpy() 操作画面内存时做过了几次，但是这些方法有一些问题。

首先，你采用主 CPU 进行内存填充，所以主总线是传输的一部分。其次，组成画面的 VRAM 不可能总是线性。这时，你不得不进行一线一线填充或者移动。但是，利用硬件图形变换，你可以马上直接填充或者移动大块 VRAM 或者 DirectDraw 画面。

DirectDraw 的两个图形变换函数是 IDirectDrawSurface4::Blt() 和 IDirectDrawSurface4::BltFast()，原型为：

```
HRESULT Blt(LPRECT lpDestRect, // dest RECT
LPDIRECTDRAW_SURFACE4 lpDDSrcSurface, // dest surface
LPRECT lpSrcRect, // source RECT
DWORD dwFlags, // control flags
LPDDBLTFX lpDDBltFx); //special fx (very cool!)
```

下面定义的参数的意义参照图 7.15。

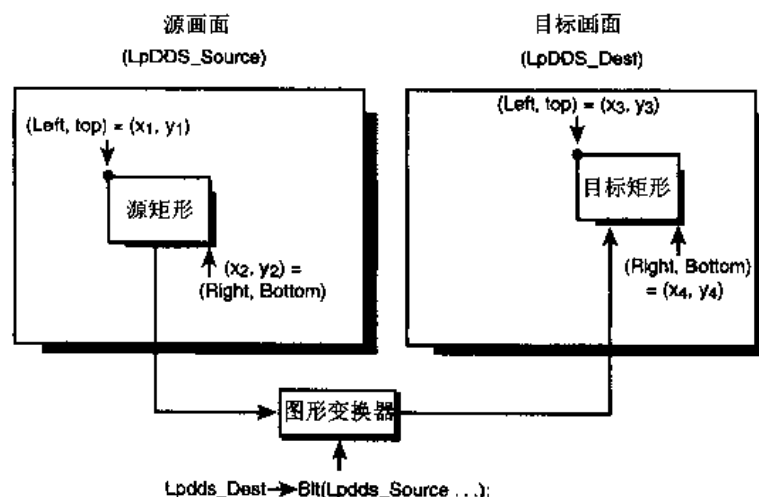


图 7.15 从源图形变换到目标

lpDestRect 定义了左上角和右下角的四边形结构，是要向目标画面图形变换的区域。如果参数为 NULL，使用整个目标画面。

lpDestSurface 用作源的 DirectDraw 画面的 IDirectDrawSurface4 接口地址。

lpSrcRect 定义了左上角和右下角的四边形结构, 是用来图形变换的源画面区域。如果参数为 NULL, 使用整个源画面。

dwFlags 决定接下来参数 DDBLTFX 结构的有效成员的个数。在 DDBLTFX 中, 特殊的行为缩放、旋转等等可以控制, 还有色彩关键字的信息。dwFlags 有效标志列在表 7.3 中。

lpDDBltFx 是一个包含有关图形变换所需要的信息的结构。数据结构如下:

```
typedef struct _DDBLTFX
{
    DWORD dwSize; // the size of this structure in bytes
    DWORD dwDDFX; // type of blitter fx
    DWORD dwROP; // Win32 raster ops that are supported
    DWORD dwDDROP; // DirectDraw raster ops that are supported
    DWORD dwRotationAngle; // angle for rotations
    DWORD dwZBufferOpCode; // z-buffer fields (advanced)
    DWORD dwZBufferLow; // advanced..
    DWORD dwZBufferHigh; // advanced..
    DWORD dwZBufferBaseDest; // advanced..
    DWORD dwZDestConstBitDepth; // advanced..
    union
    {
        {
            DWORD dwZDestConst; // advanced..
            LPDIRECTDRAW_SURFACE lpDDSZBufferDest; // advanced..
        };
        DWORD dwZSrcConstBitDepth; // advanced..
    }
    union
    {
        {
            DWORD dwZSrcConst; // advanced..
            LPDIRECTDRAW_SURFACE lpDDSZBufferSrc; // advanced..
        };
        DWORD dwAlphaEdgeBlendBitDepth; // alpha stuff (advanced)
        DWORD dwAlphaEdgeBlend; // advanced..
        DWORD dwReserved; // advanced..
        DWORD dwAlphaDestConstBitDepth; // advanced..
    }
    union
    {
        {
            DWORD dwAlphaDestConst; // advanced..
            LPDIRECTDRAW_SURFACE lpDDSAAlphaDest; // advanced..
        };
        DWORD dwAlphaSrcConstBitDepth; // advanced..
    }
    union
    {
        {
            DWORD dwAlphaSrcConst; // advanced..
            LPDIRECTDRAW_SURFACE lpDDSAAlphaSrc; // advanced..
        };
        union // these are very important
        {
            DWORD dwFillColor; // color word used for fill
            DWORD dwFillDepth; // z filling (advanced)
            DWORD dwFillPixel; // color fill word for RGB(alpha) fills
        }
    }
};
```

```

LPDIRECTDRAWSURFACE lpDDSPattern;
};
/* these are very important
DDCOLORKEY ddckDestColorkey; // destination color key
DDCOLORKEY ddckSrcColorkey; // source color key
} DDBLTFX, FAR* LPDDBLTFX;

```

注意：对有用的地方采用了黑体。

表 7.3 Bit() 的参数 dwFlags 的控制标志

值	描 述
	常用标志
DDBLT_COLORFILL	使用 DDBLTFX 结构中的 dwFillColor 成员作为填充在目标画面上目标矩形框的 RGB 颜色
DDBLT_DDFX	使用 DDBLTFX 结构中的 dwDDFX 成员指出使用该图形变换的效果
DDBLT_DDROPS	使用 DDBLTFX 结构中的 dwDDROP 成员指出非 Win32 API 部分的图形变换操作
DDBLT_DEPTHFILL	使用 DDBLTFX 结构中的 dwFillDepth 成员作为填充到目标深度缓冲画面的目标矩形框的深度值
DDBLT_KEYDESTOVERRIDE	使用 DDBLTFX 结构中的 dwckDestColorkey 成员作为目标画面的色彩关键字
DDBLT_KEYSRCOVERRIDE	使用 DDBLTFX 结构中的 dwckSrcColorkey 成员作为源画面的色彩关键字
DDBLT_ROP	在本次图形变换的图形变换操作中使用 DDBLTFX 结构中的 dwROP 成员。这些图形变换操作和 Win32 API 中定义的相同
DDBLT_ROTATIONANGLE	使用 DDBLTFX 结构中的 dwRotationAngle 成员作为该画面的旋转角 (1/100 度)。这是惟一使用硬件支持的部分，请记住 HEL (硬件仿真层) 不能进行此项工作
	色彩关键字标志
DDBLT_KEYDEST	使用和目标画面有关的色彩关键字
DDBLT_KEYSRC	使用和源画面有关的色彩关键字
	动作标志
DDBLT_ASYNC	依接收次序通过 FIFO (先进先出) 异步执行转换。若 FIFO 硬件中没有足够空间，则该调用失败。它速度快，但很危险。该标志需要合理使用出错逻辑
DDBLT_WAIT	等待直到运行该图形变换，并且如果图形变换器正忙，不返回错误信息 DDERR_WASSTILLDRAWING

注意：最有用的标志采用了黑体。

如果说你不知所措，那是瞎说——你正一步步跟随着我。现在看看函数 `BltFast()`：

```
HRESULT BltFast(
    DWORD dwx, // x-position of blit on destination
    DWORD dwy, // y position of blit on destination
    LPDIRECTDRAWSURFACE4 lpDDSrcSurface, // source surface
    LPRECT lpSrcRect, // source RECT to blit from
    DWORD dwTrans); // type of transfer
```

`dwX`, `dwY` 是在目标画面上需要图形变换的 (`x`, `y`) 坐标。

`lpDDSrcSurface` 是 DirectDraw 的源画面的 `IDIRECTDRAWSURFACE4` 接口地址。

`lpSrcRect` 是定义了左上角和右下角的四边形结构，是就用来图形变换的源画面区域。

`dwTrans` 是图形变换操作的类型，表 7.4 给出了可能的值。

表 7.4 `BltFast()` 的图形变换操作控制标志

值	描 述
DDBLTFAST_SRCCOLORFILL	指定一次透明的使用源色彩关键字的图形变换
DDBLTFAST_DESTCOLORKEY	指定一次透明的使用目标色彩关键字的图形变换
DDBLTFAST_NOCOLORKEY	指定一次不透明的标准复制的图形变换。在一些硬件上速度要快一些；在 HEL 上速度更快
DDBLTFAST_WAIT	当图形变换器正忙，并且没有返回错误信息 <code>DDERR_WASSTILLDRAWING</code> 时，强制图形变换器等待，只要运行该图形变换，或者发生一系列错误时， <code>BltFast()</code> 就返回

注意：最有用的标志采用了黑体。

好了，第一个问题是“为什么有两个不同的图形变换函数”。答案从函数本身获得：`Blt()` 是复杂选择的模型，而 `BltFast()` 是具有较少可选参数的简单模型。另外，`Blt()` 使用 DirectDraw 剪切，而 `BltFast()` 没有。这就意味着 `BltFast()` 比 `Blt()` 更快，在 HEL 时大约快 10%，硬件上甚至更快（如果硬件是在剪切上不方便）。关键点是在需要剪切的时候用 `Blt()`，不需要时用 `BltFast()`。

现在给你演示 `Blt()` 函数()填充画面的用法。这相当简单，因为没有源画面（只有目标画面）。因此许多参数都为 `NULL`。填充内存，需要执行下面几步：

1. 将要填充的色彩索引或者 RGB 码放进 `DDBLTFX` 结构的 `dwColorFill0` 字段。
2. 在你的目标画面上设计要填充的区域 `RECT` 结构。
3. 用控制标志 `DDBLT_COLORFILL` 或 `DDBLT_WAIT` 从 `IDIRECTDRAWSURFACE4` 接口指针调用 `Blt()`。这点很重要，`Blt()` 和 `BltFast()` 都是从目标画面界面上调用，而不是从源画面上调用。

下面是使用一种色彩填充 8 位画面区域的代码。

```

DDBLTFX ddbltfx; // the blitter fx structure
RECT dest_rect;  // used to hold the destination RECT
// first initialize the DDBLTFX structure
DDRAW_INIT_STRUCTURE(ddbltfx);

// now set the color word info to the color we desire
// in this case, we are assuming an 8-bit mode, hence,
// we'll use a color index from 0-255, but if this was a
// 16/24/32 bit example then we would fill the word with
// the RGB encoding for the pixel - remember!
Ddbltfx.dwFillColor = color_index;

// now set up the RECT structure to fill the region from
// (x1, y1) to (x2, y2) on the destination surface
dest_rect.left = x1;
dest_rect.top = y1;
dest_rect.right = x2;
dest_rect.bottom = y2;

// make the blitter call
lpddsprimary->Blt(&dest_rect, // pointer to dest RECT
NULL, // pointer to source surface
NULL, // pointer to source RECT
DDBLT_COLORFILL | DDBLT_WAIT,
// do a color fill and wait if you have to
&ddbltfx); // pointer to DDBLTFX holding info

```

注 意

在你将任何一个矩形结构发送给 DirectDraw 函数时，这里有一个很小的细节。一般，它们是包含左上角，除去右下角。换句话说，就是如果发送一个 RECT 是(0, 0)到(10, 10)，实际的扫描区域是(0, 0)到(9, 9)。记住这一点。如果想将 640×480 屏幕全部填充，则需将左上角设为(0, 0)，右下角设为(641, 481)。

有一重要事情需引起注意，即对源画面和矩形的设置为 NULL。这样才合理，因为这样可以使用图形变换填充色彩，而不从一个区域向另外一个区域拷贝数据。OK，下面我们继续。

前面的例子是 8 位画面。在 16/24/32 位模式下移植，只需将要填充的反映像素值的 ddbltfx.dwFillColor 改变。不是很酷吗？

例如，如果显示模式刚好是 16 位模式，你想将屏幕填充绿色，可以这样做：

```
ddbltfx.dwFillColor = _RGB16BIT565(0, 255, 0);
```

在前面 8 位下的一切其他东西都无需改变。DirectDraw 还不错吧？

为了看到图形变换硬件的效果，我写了一个演示程序 DEMO7_6.CPP1EXE。它将系统设为 640×480×16 位模式，然后用随机的色彩填充屏幕上不同的区域。你将看到每秒刷上屏幕的无数色彩（看时要关上灯）。请看下面 GameMain()，也很琐碎。

```

int Game_Main(void *parms = NULL, int num_parms = 0)
{
    // this is the main loop of the game, do all your processing
    // here

    DDBLTFX ddbltfx; // the blitter fx structure
    RECT dest_rect; // used to hold the destination RECT

    // make sure this isn't executed again
    if (window_closed)
        return(0);

    // for now test if user is hitting ESC and send WM_CLOSE
    if (KEYDOWN(VK_ESCAPE))
    {
        PostMessage(main_window_handle, WM_CLOSE, 0, 0);
        window_closed = 1;
    } // end if

    // first initialize the DDBLTFX structure
    DDRAW_INIT_STRUCT(ddbltfx);

    // now set the color word info to the color we desire
    // in this case, we are assuming an 8-bit mode, hence,
    // we'll use a color index from 0-255, but if this was a
    // 16/24/32 bit example then we would fill the WORD with
    // the RGB encoding for the pixel - remember!
    ddbltfx.dwFillColor = _RGB16BIT565(rand()%256, rand()%256, rand()%256);

    // get a random rectangle
    int x1 = rand()%SCREEN_WIDTH;
    int y1 = rand()%SCREEN_HEIGHT;
    int x2 = rand()%SCREEN_WIDTH;
    int y2 = rand()%SCREEN_HEIGHT;

    // now set up the RECT structure to fill the region from
    // (x1,y1) to (x2,y2) on the destination surface
    dest_rect.left = x1;
    dest_rect.top = y1;
    dest_rect.right = x2;
    dest_rect.bottom = y2;

    // make the blitter call
    if (FAILED(lpddsprimary->Blt(&dest_rect, // pointer to dest RECT
                                NULL, // pointer to source surface
                                NULL, // pointer to source RECT
                                DDBLT_COLORFILL | DDBLT_WAIT,
                                // do a color fill and wait if you have to
                                &ddbltfx))) // pointer to DDBLTFX holding info

```

```

        return(0);

        // return success or failure or your own return code here
        return(1);

    } // end Game_Main

```

既然现在知道了如何利用图形变换填充，让我来给你看看如何利用它来从一个画面向另外一个画面拷贝数据。这是图形变换的威力所在，也是后面的图形变换物体的基础。

从一个画面向另一个画面复制位图

图形变换就是从一些源内存向目标内存复制矩形图像。这可以包括复制整个屏幕，或者在游戏中代表小物体的小图像。两种情况下，都需要引导图形变换，从一个画面向另外一个复制数据。实际上，你已经知道如何做了，只是还没有意识到。图形变换填充演示程序需要两处改动。

当时用 Blt()函数时，你一般发送一个源矩形和画面以及一个目标矩形和画面执行图形变换。图形变换将从源矩形向目标矩形拷贝像素。源和目标画面可以是同一个（画面向画面拷贝或移动），但是通常是不同的。一般，后者是多数精灵引擎的基础（精灵是游戏中在屏幕上到处移动的小物体）。

现在，你知道了如何创建一个主画面和一个后备缓冲的备用画面。但是，你不知道如何创建同主画面相联系的备用平面。如果不能够创建它们，就不能够图形变换它们。因此，直到我教给你从后备缓冲向主画面图形变换后，我才演示一些画面向主画面的图形变换。而后，从普通画面向主画面或者后备缓冲的图形变换就简单了。

要从任意两个画面图形变换，所有要做的就是正确设置矩形并用正确的参数调用 Blt()。见图 7.15。想像你要从(x1, y1)(x2, y2)定义的矩形区域（本例中是后备缓冲）向(x3, y3)(x4, y4)的目标画面（本例中是主画面）拷贝。下面是其代码：

```

RECT source_rect;    // used to hold source RECT
dest_rect;           // used to hold the destination RECT

// now set up the RECT structure to fill the region from
// (x1, y1) to (x2, y2) on the destination surface
dest_rect.left = x1;
dest_rect.top = y1;
dest_rect.right = x2;
dest_rect.bottom = y2;

// now set up the RECT structure to fill the region from
// (x3, y3) to (x4, y4) on the destination surface
dest_rect.left = x3;
dest_rect.top = y3;

```



```

dest_rect.right = x4;
dest_rect.bottom = y4;

// make the blitter call
lpddsprimary->Blt(&dest_rect, // pointer to dest RECT
NULL, // pointer to source surface
NULL, // pointer to source RECT
DDBLT_COLORFILL, DDBLT_WAIT,
// do a color fill and wait if you have to
&ddbltfx); // pointer to DDBLTFX holding info

```

哈哈!很简单,当然,我还留下了一些细节,如透明、剪切。我先来谈谈剪切。图 7.16 描述了画上画面的经过和未经过剪切的位图。如果位图超出了目标画面的范围,不经剪切就图形变换显然有问题,会出现内存可能被重写等情况。所以 DirectDraw 支持通过 IDirectDrawClipper 接口进行的剪切。或者,如果你要写自己的光栅化位图程序,像在 Blit8×8()中那样,你总能够添加剪切代码。但是,那样会使事情变慢。下面我们谈谈图形变换透明性。

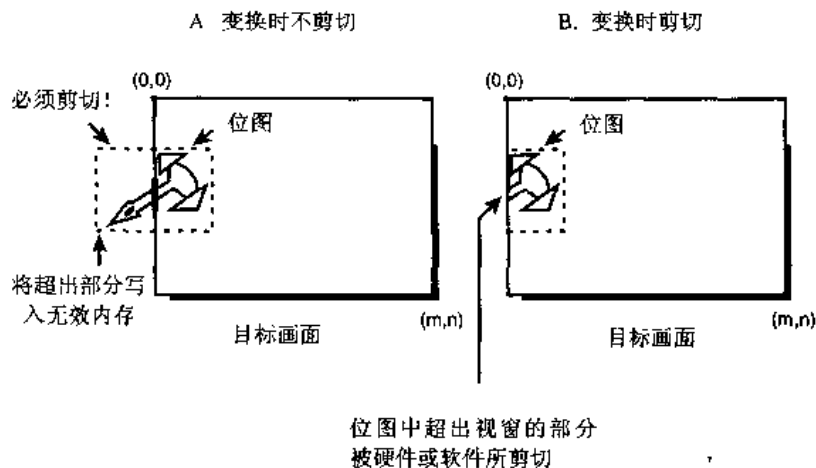


图 7.16 基本位图剪切问题

当你画一个位图时,位图总是在一个像素的矩形矩阵中。但是,当进行图形变换时,你并不希望所有的像素都被拷贝。许多时候,你会选择一个色彩作为透明色彩,如:黑色、蓝色、绿色等等,它们不被拷贝(参见 Blit8×8())。DirectDraw 支持称为色彩开关的技术,我在下面简单地谈谈。

在移到剪切之前,我想给你一个从后备缓冲向主画面图形变换的演示程序。看 CD 中的 DEMO7_7.CPP1EXE。惟一问题是我还没有教给你如何从磁盘装载位图。所以我现在还不能图形变换一些很酷的位图。因此我在后备缓冲上画了一个从顶到底具有梯度的 16 位模式的绿色,用它作为源数据。你将看到一些梯度矩形以一定速度拷贝到主画面。下面是你看到的程序的 GameMain():

```

int Game_Main(void *parms = NULL, int num_parms = 0)
{
    // this is the main loop of the game, do all your processing
    // here

    RECT source_rect, // used to hold the destination RECT
        dest_rect; // used to hold the destination RECT

    // make sure this isn't executed again
    if (window_closed)
        return(0);

    // for now test if user is hitting ESC and send WM_CLOSE
    if (KEYDOWN(VK_ESCAPE))
    {
        PostMessage(main_window_handle, WM_CLOSE, 0, 0);
        window_closed = 1;
    } // end if

    // get a random rectangle for source
    int x1 = rand()%SCREEN_WIDTH;
    int y1 = rand()%SCREEN_HEIGHT;
    int x2 = rand()%SCREEN_WIDTH;
    int y2 = rand()%SCREEN_HEIGHT;

    // get a random rectangle for destination
    int x3 = rand()%SCREEN_WIDTH;
    int y3 = rand()%SCREEN_HEIGHT;
    int x4 = rand()%SCREEN_WIDTH;
    int y4 = rand()%SCREEN_HEIGHT;

    // now set up the RECT structure to fill the region from
    // (x1,y1) to (x2,y2) on the source surface
    source_rect.left   = x1;
    source_rect.top    = y1;
    source_rect.right  = x2;
    source_rect.bottom = y2;

    // now set up the RECT structure to fill the region from
    // (x3,y3) to (x4,y4) on the destination surface
    dest_rect.left    = x3;
    dest_rect.top     = y3;
    dest_rect.right   = x4;
    dest_rect.bottom  = y4;

    // make the blitter call
    if (FAILED(lpddsprimary->Blt(&dest_rect, // pointer to dest RECT
                                lpddsback,   // pointer to source surface

```

```

        &source_rect, // pointer to source RECT
        DDRT_WAIT,    // control flags
        NULL)))       // pointer to DDBLTFX holding info
    return(0);

    // return success or failure or your own return code here
    return(1);

} // end Game_Main

```

在 `Game_Init()` 中，我用了一个内联汇编语言完成 `DWORD` 或者两个 16 位像素的 32 位直线绘制，以 `RGB.RGB` 格式，而不是 8 位填充格式。下面是代码。

```

_asm
{
    CLD                      ; // clear direction of copy to forward
    MOV EAX, color           ; // color goes here
    MOV ECX, (SCREEN_WIDTH/2); // number of DWORDS goes here
    MOV EDI, video_buffer    ; // address of line to move data
    REP STOSD                ; // color goes here
} // end asm

```

基本上，前面的汇编语言完成下面的 C++ 循环。

```

for (DWORD ecx = 0, DWORD *edi = video_buffer;
    ecx < (SCREEN_WIDTH/2); ecx++)
    edi[ecx] = color;

```

如果不知道汇编语言，别泄气，我不过喜欢用它在这或那做一些小的事情。它也是用内联汇编语言的很好的练习。

作为练习，看看你是否能让程序只在主画面上工作。把后备缓冲去掉，在主画面上绘图，将源和目标画面作为一个，运行图形变换。观察发生的事情……

剪切基础

本书中我将一遍遍讲到剪切。像素剪切、位图剪切、2D 剪切、3D 剪切，甚至更多。不过，现在的主题是 `DirectDraw`。我将给你集中讲解像素剪切和位图剪切，帮你轻松过渡到物体剪切上，可以向你保证，3D 剪切会非常复杂。

剪切通常定义为：“不画在视区或窗口之外的像素或图像部分。”就像 Windows 将剪切画向你的窗口的用户区的任何东西一样，你需要对运行在 `DirectX` 上的游戏这样做。

现在，就像二维位图一样，`DirectDraw` 做的仅仅是加速位图或者位图形变换。当然，

许多卡支持画线、圆和其他一些二次曲线，但是 DirectDraw 不支持，所以不要利用它们（虽然但愿你不久以后能够使用）。

所有这些意味着，如果你想写一个位图引擎画像素、直线、位图，需要你自己对画的线、点进行剪切。但是，DirectDraw 可以帮助剪切位图，条件是位图正好是 DirectDraw 的画面模式，或者 IDirectDrawSurfaces 模式。

DirectDraw 提供的帮助是 IDirectDrawSurfaces 接口下的 DirectDraw 剪切器。需要你做的事是创建一个 IDirectDrawClipper，给它有效的剪切区域，然后将它同画面连接。这样，当你用图形变换函数 Blt() 时，它将剪切到剪切区，如果你有适当的硬件的话，你就无需再做任何其他事情了。但是，首先看看如何剪切像素和创建一个 Blt8×8() 剪切函数。

将像素剪切到视区

图 7.17 给出了图解。假设剪切一个位于 (x, y) 的像素到视区 $(x1, y1)(x2, y2)$ 。如果 (x, y) 在 $(x1, y1)(x2, y2)$ 定义的四边形之内，对它进行渲染；否则不渲染，够简单吧？

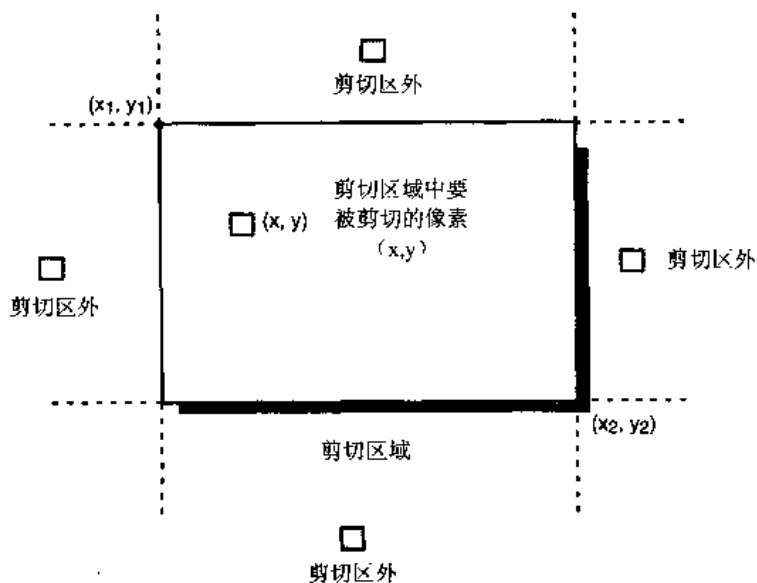


图 7.17 剪切区域的详细内容

下面是适用于 640×480 线性 8 位模式的代码。

```
// assume clipping rectangle is global
int x1, y1, x2, y2; // these are defined somewhere

void Plot_Pixel_Clp8(int x, int y,
    UCHAR color,
    UCHAR *video_buffer)
{

```

```
// test the pixel to see if it's in range
if (x>=x1 && x<=x2 && y>=y1 && y<=y2)
video_buffer[x+y*640] = color;
} // end if
```

当然，有许多优化的空间。但是要领会主旨，你已经创建了一个像素软件过滤器，只有满足 if 语句的像素坐标才能通过过滤器，这真是一个有趣的概念。现在，前面的剪切非常通用，但在许多时候，窗口或视区在(0, 0)处，具有一定尺寸(win_width, win_height)。这样就可以将代码简化一点。

```
// assume clipping rectangle is global
int x1, y1, x2, y2; // these are defined somewhere

void Plot_Pixel2_Clp8(int x, int y,
    UCHAR color,
    UCHAR *video_buffer)
{
    // test the pixel to see if it's in range
    if (x>=0 && x<win_width && y>=0 && y<=win_height)
        video_buffer[x+y*640] = color;
} // end if
```

看到了吗？另外，无论何时，有 0 时，可以做更进一步的优化。现在知道了剪切的基本点，并知道如何去做，下面我将教你如何剪切一个位图。

以硬拷贝的方式剪切位图

剪切位图同剪切像素一样简单。有两种方法。

方法 1：就像生成像素一样自由剪切位图中的每一个像素，这样做方法简单，但是速度慢。

方法 2：剪切位图的矩形边界到视区中，画位于视区中的位图部分。较复杂，但是快，几乎不影响执行，不会冲击内部循环。

显然，你会采用方法 2，如图 7.18 所示。我先进行一点假设，假设屏幕从(0, 0)到(SCREEN_WIDTH-1, SCREEN_HIGHT-1)，你的位图左上角为(x, y)，正好具有同尺寸一样大小的像素数，换句话说，就是从(x, y)到(x+width-1, y+high-1)。请花费一点时间搞明白-1 的理由。一般，如果图像是 1×1 图像，则它的高度和宽度分别为 1，因此，如果它的起始点在(x, y)，位图为从(x, y)到(x+1-1, y+1-1)即(x, y)。这是因为当只有一个像素时，需要“-1”。

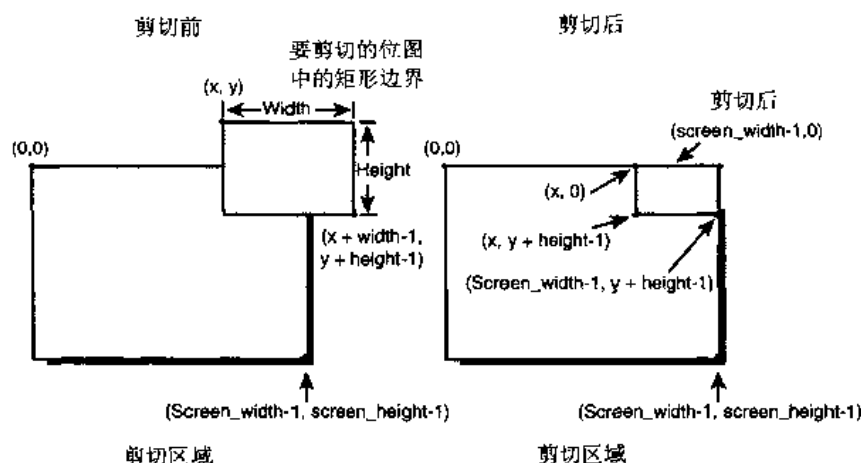


图 7.18 如何剪切位图的边界框

剪切的计划很简单，你只需剪切位图在视区中的可见四边形，然后画出在被剪切位图中的可见部分，下面是 $640 \times 480 \times 8$ 线性模式下的代码。

```
// dimensions of window or viewport (0,0) is origin
#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480

void Blit_Clippped(int x, int y,          // position to draw bitmap
                  int width, int height, // size of bitmap in pixels
                  UCHAR *bitmap,         // pointer to bitmap data
                  UCHAR *video_buffer)   // pointer to video buffer surface
{
    // this function blits and clips the image sent in bitmap to the
    // destination surface pointed to by video_buffer
    // the function assumes a 640x480x8 mode with linear pitch

    // first do trivial rejections of bitmap, is it totally invisible?
    if ((x >= SCREEN_WIDTH) || (y >= SCREEN_HEIGHT) ||
        ((x + width) <= 0) || ((y + height) <= 0))
        return;

    // clip source rectangle
    // pre-compute the bounding rect to make life easy
    int x1 = x;
    int y1 = y;
    int x2 = x1 + width - 1;
    int y2 = y1 + height - 1;

    // upper left hand corner first
    if (x1 < 0)
```

```

    x1 = 0;

    if (y1 < 0)
        y1 = 0;

    // now lower left hand corner
    if (x2 >= SCREEN_WIDTH)
        x2 = SCREEN_WIDTH-1;

    if (y2 >= SCREEN_HEIGHT)
        y2 = SCREEN_HEIGHT-1;

    // now we know to draw only the portions
    // of the bitmap from (x1,y1) to (x2,y2)
    // compute offsets into bitmap on x,y axes,
    // we need this to compute starting point
    // to rasterize from
    int x_off = x1 - x;
    int y_off = y1 - y;

    // compute number of columns and rows to blit
    int dx = x2 - x1 + 1;
    int dy = y2 - y1 + 1;

    // compute starting address in video_buffer
    video_buffer += (x1 + y1*640);

    // compute starting address in bitmap to scan data from
    bitmap += (x_off + y_off*width);

    // at this point bitmap is pointing to the first
    // pixel in the bitmap that needs to
    // be blitted, and video_buffer is pointing to
    // the memory location on the destination
    // buffer to put it, so now enter rasterizer loop

    UCHAR pixel; // used to read/write pixels

    for (int index_y = 0; index_y < dy; index_y++)
    {
        // inner loop, where the action takes place
        for (int index_x = 0; index_x < dx; index_x++)
        {
            // read pixel from source bitmap,
            // test for transparency and plot
            if ((pixel = bitmap[index_x]))
                video_buffer[index_x] = pixel;
        }
        // end for index_x
    }

```

```

        // advance pointers
        video_buffer += 640; // bytes per scanline
        bitmap        += width; // bytes per bitmap row

    } // end for index_y

} // end Blit_Clippped

```

作为一个剪切的演示软件。我已经写出了你所看到的最原始的位图引擎。首先，我创建一个 64 字节的数组，一个表示笑脸的数组。下面是初始化语句。

```

UCHAR happy_bitmap[64] = {0, 0, 0, 0, 0, 0, 0, 0,
                           0, 0, 1, 1, 1, 1, 0, 0,
                           0, 1, 0, 1, 1, 0, 1, 0,
                           0, 1, 1, 1, 1, 1, 1, 0,
                           0, 1, 0, 1, 1, 0, 1, 0,
                           0, 1, 1, 0, 0, 1, 1, 0,
                           0, 0, 1, 1, 1, 1, 0, 0,
                           0, 0, 0, 0, 0, 0, 0, 0};

```

然后，我把系统设为 $320 \times 240 \times 8$ 模式，色彩索引为 RGB(250, 250, 0)，就是黄色。我让笑脸以一个随机速度移动。当它距离屏幕四周边缘任何一边太远时，擦掉笑脸。它可以跑出窗口很远，以便你看清函数的工作过程。总共产生并擦除 100 个笑脸，最终程序见 DEMO7_8.CPP1EXE。图 7.19 是程序运行时的其中一个屏幕镜头。

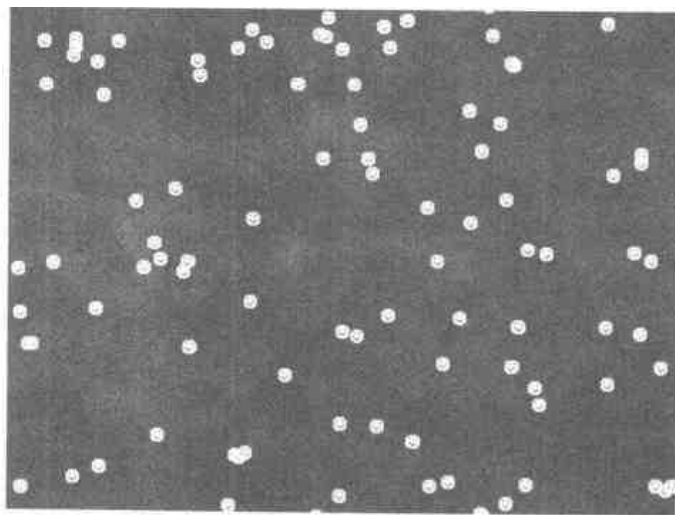


图 7.19 运行中的 DEMO8.EXE 程序

下面是程序的 GameMain():

```

int Game_Main(void *parms = NULL, int num_parms = 0)
{

```



```
// this is the main loop of the game, do all your processing
// here

DDBLTFX ddbltfx; // the blitter fx structure

// make sure this isn't executed again
if (window_closed)
    return(0);

// for now test if user is hitting ESC and send WM_CLOSE
if (KEYDOWN(VK_ESCAPE))
{
    PostMessage(main_window_handle, WM_CLOSE, 0, 0);
    window_closed = 1;
} // end if

// use the blitter to erase the back buffer
// first initialize the DDBLTFX structure
DDRAW_INIT_STRUCT(ddbltfx);

// now set the color word info to the color we desire
ddbltfx.dwFillColor = 0;

// make the blitter call
if (FAILED(lpddsback->Blt(NULL, // ptr to dest RECT, NULL means all
                        NULL, // pointer to source surface
                        NULL, // pointer to source RECT
                        DDBLT_COLORFILL | DDBLT_WAIT,
                        // do a color fill and wait if you have to
                        &ddbltfx))) // pointer to DDBLTFX holding info
    return(0);

// initialize ddsd
DDRAW_INIT_STRUCT(ddsds);

// lock the back buffer surface
if (FAILED(lpddsback->Lock(NULL, &ddsds,
                        DDLOCK_WAIT | DDLOCK_SURFACEMEMORYPTR,
                        NULL)))
    return(0);

// draw all the happy faces
for (int face=0; face < 100; face++)
{
    Blit_Clippped(happy_faces[face].x,
                  happy_faces[face].y,
                  8, 8,
                  happy_bitmap,
```

```

        (UCHAR *)ddsd.lpSurface,
        ddsd.lPitch);
    } // end face

// move all happy faces
for (face=0; face < 100; face++)
{
    // move
    happy_faces[face].x+=happy_faces[face].xv;
    happy_faces[face].y+=happy_faces[face].yv;

    // check for off screen, if so wrap
    if (happy_faces[face].x > SCREEN_WIDTH)
        happy_faces[face].x = -8;
    else
    if (happy_faces[face].x < -8)
        happy_faces[face].x = SCREEN_WIDTH;

    if (happy_faces[face].y > SCREEN_HEIGHT)
        happy_faces[face].y = -8;
    else
    if (happy_faces[face].y < -8)
        happy_faces[face].y = SCREEN_HEIGHT;

    } // end face

// unlock surface
if (FAILED(lpddsback->Unlock(NULL)))
    return(0);

// flip the pages
while (FAILED(lpddsprimary->Flip(NULL, DDFLIP_WAIT)));

// wait a sec
Sleep(30);

// return success or failure or your own return code here
return(1);

} // end Game_Main

```

注意



在演示程序中一定要看 Blit_Clippped()函数的代码。因为我将它改变为一个采用变内存步长工作的函数。这不是大问题，你也可能会问为什么采用“320×240”模式，这是因为，8×8的位图在640×480模式下太小。

使用 IDirectDrawClipper 进行 DirectDraw 剪切

你已经看到了如何利用软件进行剪切，现在来看看用 DirectDraw 剪切是多么简单。DirectDraw 有一个称作为 IDirectDrawClipper 的接口用于所有的 2D 图形变换剪切，就像在 DirectX3D 下，进行 3D 着色一样。但是现在，你只是对使用剪切板剪切有函数 Blt()绘制的位图以及相关的硬件有兴趣。

设置 DirectDraw 剪切，需要做下面这些。

1. 创建 DirectDraw 剪切板。
2. 创建剪切列表。
3. 用 IDirectDrawClipper::SetClipList()将剪切列表发送给剪切板。
4. 用 IDirectDrawSurface4::SetClipper()将剪切板同窗口或者画面连接。

下面开始第一步。函数创建了一个 IDirectDrawClipper 接口，称为 IDirectDraw4::CreateClipper(), 如下所示。

```
HRESULT CreateClipper(DWORD dwFlags, // control flags
LPDIRECTDRAWCLIPPER FAR *lplpDDClipper, // address of interface pointer
IUnknown FAR *pUnkOuter); // COM stuff
```

如果成功，则函数返回 DD_OK。

参数相当简单。dwFlags 现在没有用，必须置为 0，lplpDDClipper 是 IDirectDrawClipper 接口的地址，函数成功后指向一个 DirectDraw 剪切板。pUnkOuter 和 COM 有关，你不必知道，将它设为 NULL。创建一个剪切板对象，输入如下代码。

```
LPDIRECTDRAWCLIPPER lpDDClipper = NULL; // hold the clipper
if (FAILED(lpdd->CreateClipper(0, &lpddclipper, NULL)))
return(0);
```

如果函数成功，lpddClipper 指向一个有效的 IDirectDrawClipper 接口，你可以在它上面调用线程。

很好，但你如何创建剪切序列，它有代表什么？在 DirectDraw 下，剪切列表给出一个存储 RECT 结构的序列。如图 7.20 所示，表示可以图形变换的有效区域。如你所见，在显示画面有许多矩形，但是 DirectDraw 图形变换系统只能够图形变换这些区域。你可以通过锁定/解锁在任何地方绘制位图，但是，图形变换硬件只能够图形变换剪切区域，通常称之为剪切序列。

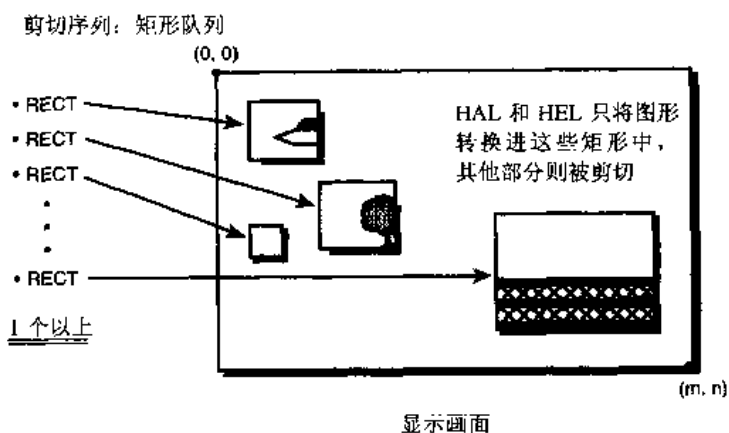


图 7.20 剪切序列与图形变换的关系

创建一个剪切序列，只需填写称为 **RGNDATA** 的数据结构。如下所示：

```
typedef struct _RGNDATA
{
    /* rgnd */
    RGNDATAHEADER rdh; // header info
    Char Buffer[1]; // the actual RECT list
} RGNDATA;
```

这是非常古怪的数据结构，它一般是可变大小的结构，这意味着 **Buffer[]** 部分可以是任何长度。结构动态产生而非静态。它的实际长度存储在 **RGNDATAHEADER** 中。下面你看到的是一个新的 **DirectX** 数据结构的版本，设置每个结构的 **dwSize** 字段。也许将 **Buffer[]** 设成一个指针比存储单个字节更好。

不管如何认为，这是一个事实：需要你做的一切就是给 **RGNDATAHEADER** 结构分配足够的内存，在内存中同时连续存储一个或者多个矩形结构数组，如图 7.21 所示。将它变为 **RGNDATA** 类型并通过它。

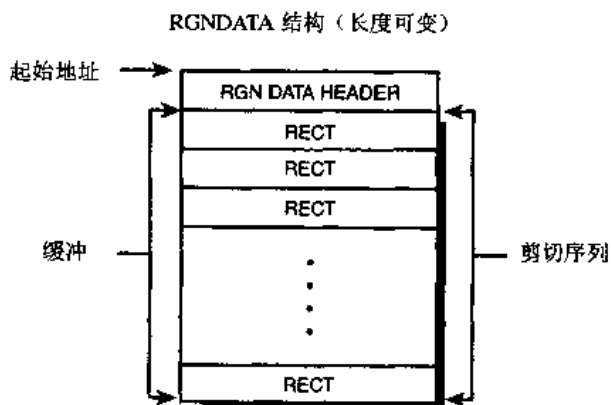


图 7.21 RGNDATA 剪切结构的内存分布

看看在 RGNDATAHEADER 结构中都是些什么：

```
typedef struct _RGNDATA
{ // rgndh
  DWORD dwSize; // size of this header in bytes
  DWORD iType; // type of region data
  DWORD nCount; // number of RECT's in Buffer[]
  DWORD nRgnSize; // size of Buffer[]
  RECT rcBound; // a bounding box around all RECTS
} RGNDATA;
```

建立这个结构，将 dwSize 设置为 sizeof(RGNDATAHEADER)，将 iType 设定为 RDH_RECTANGLES，将 nCount 设定为你的剪切序列中 RECT 矩形的数目。设 nRgnSize 为以字节为单位的 buffer[] 的大小（等于 sizeof(RECT)*nCount）。在 RECT 周围创建一个边界箱，将它存储在 rcBound 中。一旦产生了 RGNDATA 结构，就可以通过调用 IDIRECTDRAWCLIPPER::SetClipList() 发送你的剪切板了。如下所示：

```
HRESULT SetClipList(LPRGNDATA lpClipList, // ptr to RGNDATA
  DWORD dwFlags); // flags, always 0
```

关于这个要谈的不多。假设你已经为剪切序列生成了一个 RGNDATA 结构，下面给出如何设置剪切序列：

```
if (FAILED(lpddclipper->SetClipList(&rgndata, 0));
return(0);
```

一旦设置完剪切序列，你就可以最终用 IDIRECTDRAWSURFACE4::SetClipper() 将剪切板同你想要连接的画面连接。如下所示：

```
HRESULT SetClipper(LPDIRECTDRAWCLIPPER lpDDClipper);
```

下面是函数的用法：

```
if (FAILED(lpddclipper->SetClipper(&lpddclipper)))
return(0);
```

多数时候，lpddsurface 是你的备用渲染画面，如后备缓冲画面。通常，你没有将剪切板连接主画面。

好，我知道你一定迷惑得脸色发紫，因为我一直没有将创建和设置 RGNDATA 数据结构的细节告诉你。原因是太难详细揭示其细节了，直接看代码更简单一些。因此，我创建了一个剪切板和一个剪切序列在函数 Ddraw_Attach_Clipper() 中，将它们同任意画面连接。下面是代码：

```
LPDIRECTDRAWCLIPPER DDraw_Attach_Clipper(LPDIRECTDRAWSURFACE4 lpdds,
```

```

        int num_rects,
        LPRECT clip_list)

{
    // this function creates a clipper from the sent clip list and attaches
    // it to the sent surface

    int index;                // looping var
    LPDIRECTDRAWCLIPPER lpddclipper; // pointer to the newly
                                // created dd clipper
    LPRGNDATA region_data;    // pointer to the region
                                // data that contains
                                // the header and clip list

    // first create the direct draw clipper
    if (FAILED(lpdd->CreateClipper(0,&lpddclipper,NULL)))
        return(NULL);

    // now create the clip list from the sent data

    // first allocate memory for region data
    region_data = (LPRGNDATA)malloc(sizeof(RGNDATAHEADER)+
        num_rects*sizeof(RECT));

    // now copy the rects into region data
    memcpy(region_data->Buffer, clip_list, sizeof(RECT)*num_rects);

    // set up fields of header
    region_data->rdh.dwSize      = sizeof(RGNDATAHEADER);
    region_data->rdh.iType       = RDH_RECTANGLES;
    region_data->rdh.nCount      = num_rects;
    region_data->rdh.nRgnSize    = num_rects*sizeof(RECT);

    region_data->rdh.rcBound.left   = 64000;
    region_data->rdh.rcBound.top    = 64000;
    region_data->rdh.rcBound.right  = -64000;
    region_data->rdh.rcBound.bottom = -64000;

    // find bounds of all clipping regions
    for (index=0; index<num_rects; index++)
    {
        // test if the next rectangle unioned with
        // the current bound is larger
        if (clip_list[index].left < region_data->rdh.rcBound.left)
            region_data->rdh.rcBound.left = clip_list[index].left;

        if (clip_list[index].right > region_data->rdh.rcBound.right)
            region_data->rdh.rcBound.right = clip_list[index].right;

        if (clip_list[index].top < region_data->rdh.rcBound.top)

```

```

    region_data->rdh.rcBound.top = clip_list[index].top;

    if (clip_list[index].bottom > region_data->rdh.rcBound.bottom)
        region_data->rdh.rcBound.bottom = clip_list[index].bottom;

    } // end for index

// now we have computed the bounding rectangle region and set up the data
// now let's set the clipping list

if (FAILED(lpddclipper->SetClipList(region_data, 0)))
{
    // release memory and return error
    free(region_data);
    return(NULL);
} // end if

// now attach the clipper to the surface
if (FAILED(lpdds->SetClipper(lpddclipper)))
{
    // release memory and return error
    free(region_data);
    return(NULL);
} // end if

// all is well, so release memory and
// send back the pointer to the new clipper
free(region_data);
return(lpddclipper);

} // end DDraw_Attach_Clipper

```

函数用法简单。假设你有一个动画系统，系统有个称为 `lpddsprimary` 的主画面和一个称为 `lpddsback` 的备用后备缓冲，你想将一个具有下列 `RECT` 序列的剪切板连接到它上面。

```

RECT rect_list[3] = {{10, 10, 50, 50},
{100, 100, 200, 200},
{300, 300, 500, 450}};

```

如下调用它：

```

LPDIRECTDRAWCLIPPER lpDDClipper =
DDraw_Attach_Clipper(lpddsback, 3, rect_list);

```

非常好！如果进行此调用，只有在矩形(10, 10)(50, 50), (100, 100)(200, 200), (300, 300)(500, 450)中的位图部分才可见。这个函数是我写本章时所用的一个库的一部分。后面，我将给出你其中的所有函数，使你不需要自己写无聊的 `DirectDraw` 代码，可以集中精力编

程。

不管怎样，基于前面的代码，我已经创建了一个演示程序，叫做 DEMO7_9.CPPIEXE。一般，我利用图形变换演示程序 DEMO7_7.CPP，把它变成 8 位色彩模式。加入剪切板函数，使图形变换只是在显示的主画面的当前剪切区域内。另外，剪切区域是上一段中的同一序列剪切区。图 7.22 给出了程序运行时的一个画面。注意它就像是一束剪切板允许位图着色的小窗口。

下面是设置剪切板的 DEMO7_9.CPP 中的 Game_Main()代码：

```
// now create and attach clipper
RECT rect_list[3]={10, 10, 50, 50},
{100, 100, 200, 200},
{300, 300, 500, 450}};
if (FAILED(lpddclipper = DDRAW_Attach_Clipper(lpddsprimary, 3, rect_list)))
    return(0);
```

酷!现在，我已经讨厌了采用梯度色彩填充的矩形了。如果再看不到一些位图，我就会发疯。下面，我将教你给窗口装载位图。

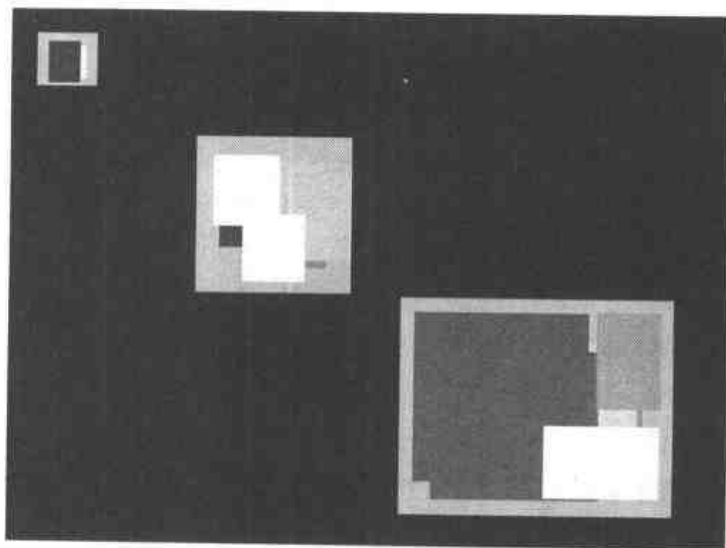


图 7.22 DEMO7_9.EXE 画面

采用位图

有很多不同的位图文件格式，但是我只是使用很少一部分进行游戏编程：PCX(PC 画)、TGA (Targa)、BMP (Windows 本身的格式)。它们各有优缺点，但你在 Windows 下工作，所以最好采用 Windows 本身的格式——.BMP，让日子好过一些。(我已经陷入了 DirectX API 的修订地狱，所以我对这点也不确定，如果我看到一个更商业化的 Don Lapre，我会马上去

邮局。)

其他格式工作原理相通,所以,如果你知道了处理一种文件格式,另外一种除了获得文件头信息,从磁盘上读一些字节之外没有什么不同。

装载 .BMP 文件

读 .BMP 文件有多种方法,你可以自己写一个读软件,也可以用 Win32 API 函数,或者两者结合使用。由于弄清 Win32 函数同你自己编写一个的难度差不多,你可能更愿意写一个 .BMP 装载文件。如图 7.23 所示, .BMP 文件包含三部分。

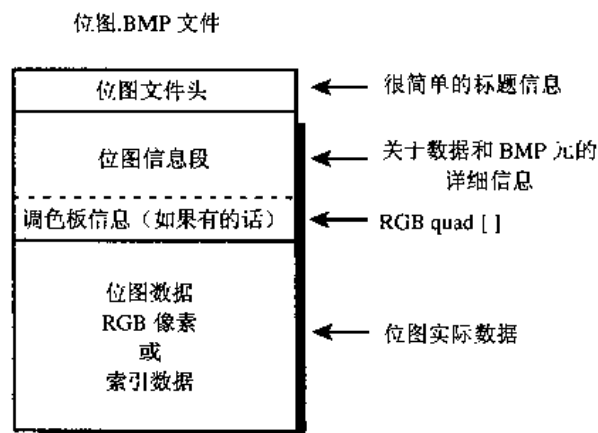


图 7.23 磁盘中 .BMP 文件的结构

三部分分别如下:

位图文件头: 它存放着图像的一般信息,存放在 Win32 的结构 BITMAPFILEHEADER 中。

```
typedef struct tagBITMAPFILEHEADER
{ // bmfh
WORD bfType; // Specifies the file type
// Must be 0x4D42 for .BMP
DWORD bfSize; // Specifies the size in bytes of
// the bitmap file
WORD bfReserved1; // Reserved; must be zero
WORD bfReserved2; // Reserved; must be zero
DWORD bfOffBits; // Specifies the offset, in
// byte, from the
// BITMAPFILEHEADER structure
// to the bitmap data
} BITMAPFILEHEADER;
```

位图信息段: 包括两部分数据结构,即 BITMAPINFOHEADER 部分和调色板信息部分

(如果有的话)。

```
typedef struct tagBITMAPINFO
{ // bmi
  BITMAPFILEHEADER bmiHeader; // the info header
  RGBQUAD bmiColors[1]; // palette(if there is one)
} BITMAPINFO;
```

下面是 BITMAPINFOHEADER 结构:

```
typedef struct tagBITMAPINFOHEADER { // bmih
  DWORD biSize; // Specifies the number of
  // bytes required by the structure
  LONG biWidth; // Specifies the width of the bitmap in pixels.
  LONG biHeight; // Specifies the height of the bitmap in pixels.
  // If biHeight is positive, the bitmap is a
  // bottom-up DIB and its
  // origin is the lower left corner
  // If biHeight is negative, the bitmap
  // is a top-down DIB and its origin is the upper left corner
  WORD biPlanes; // Specifies the number of color planes must be 1
  WORD biBitCount; // Specifies the number of bits per pixel
  // this value must be 1, 4, 8, 16, 24, or 32
  DWORD biCompression; // Specifies type of compression(advanced)
  // it will always be
  // BI_RGB for uncompressed .BMPs
  // which is what we're going to use
  DWORD biSizeImage; // size of image in bytes
  LONG biXPelsPerMeter; // Specifies the number of
  // pixels per meter in X-axis
  LONG biYPelsPerMeter; // Specifies the number of
  // pixels per meter in Y-axis
  DWORD biClrUsed; // Specifies the number of
  // colors used by the bitmap
  DWORD biClrImportant; // Specifies the number of
  // colors that are important
} BITMAPINFOHEADER;
```

注 意

8 位图像通常使 biClrUsed 和 biClrImportant 两字段为 256, 而 16 位或 24 位图像设它们为 0。因此, 通常通过测试 biBitCount 找出每像素多少位。

图像数据区: 它是一个以字节为单位的, 描述 1、4、8、16、或 24 位图像像素的数据流 (可能是压缩或非压缩格式)。数据是逐行排列的, 但是有时会颠倒过来, 数据的第一行是图像的最后一行。如图 7.24 所示。你可以通过访问符号 biHeight 来查询, 正值表示颠倒, 负值表示正常。

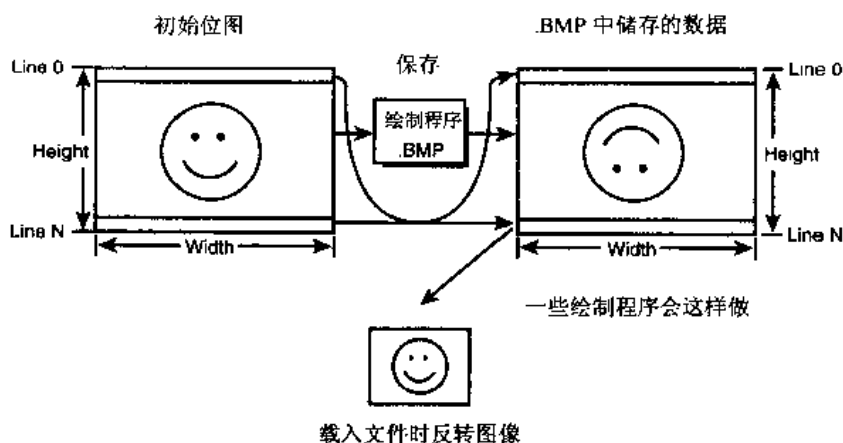


图 7.24 .BMP 文件中的图像数据在 y 轴方向有时是反向的

人工读一个.BMP 文件，首先需要打开它（你可以用喜欢的任何 I/O 技术），然后，读 BITMAPINFOHEADER。接着，读 BITMAPINFO 部分（如果是 265 色的话），它才是 BITMAPINFOHEADER 的调色板，或者你仅仅读取 BITMAPINFOHEADER 信息。由此你可以得出图像的大小（biHeight, biWidth）、色彩深度（biBitCount, biClrUsed），另外还可以读出同调色板（如果有的话）在一起的图像数据。当然，还有许多细节，如分配内存读数据、移动文件指针等。调色板也可能完全是 RGBQUAD，它是正常的 PALETTEENTRY 的相反顺序，你还需要像下面这样转换它们。

```
typedef struct tagRGBQUAD
{ // rgbq
  BYTE rgbBLUE;    // blue
  BYTE rgbGreen;   // green
  BYTE rgbRed;     // red
  BYTE rgbReserved; // unused
} RGBQUAD;
```

回到第四章“WindowsGDI、控件和突发奇想”，你可能会记起用 LoadBitmap()函数从磁盘中调位图，你可以用这个函数，但是你不能不将你所有的位图调进 EXE 文件作为资源。虽然这对整个程序来说很酷，但当你想发展的时候，它却毫无用处。一般，可以将你的图形用一个画笔或者程序进行扭曲，将它们放在一个目录下面，然后运行你的程序看看发生了什么。所以，你需要基于位图文件读取的函数，下面就要谈到。在你这样做之前，看看 Win32 API 的调图函数。运行 LoadImage()：

```
HANDLE LoadImage(
  HINSTANCE hinst, // handle of the instance that contains the image
  LPCTSTR lpszName, // name or identifier of image
  UINT uType,      // type of image
  int cxDesired,   // desired width
```

```
int cyDesired,      // desired height
UINT fuLoad);      // load flags
```

函数相当通用，但是，你只想用它从磁盘调.BMP 文件，所以你不得不为它做的其他事担心。从磁盘读.bmp 文件时将参数如下设置：

hinst——这是事件句柄，设为 NULL。

lpszName——磁盘上.BMP 文件名。给它一个以 NULL 结尾的标准.BMP 文件。如 ANDER.BMP、C:/INAGES/SHIP.BAP 等等。

uType——这是调用文件的类型。设为 IMAGE_BITMAP。

cxDesired, cyDesired——它们描述了位图的高度和宽度。如果它们不是 0，函数会自动缩放图形填充。因此，如果你知道图像的尺寸，就按已知尺寸设置。否则就设为 0，等以后读出图像的大小。

fuLoad——控制标志。设为 (LR_LOADFROMFILE|LR_CREATEDIBSECTION)。它引导 LoadImage()用已知名字 lpszName 从磁盘上读数据，但不将位图数据传送给当前显示设备。

此函数的问题在于它太通用了，获得一点数据很困难。你不得不用其他函数读文件头信息，如果有调色板，会更麻烦。所以，我建立了我自己的 Load_Bitmap_File()函数，可以从盘上调入任何格式的（包括有调色板）文件，并把其信息装入这个结构中。

```
typedef struct BITMAP_FILE_TAG
{
    BITMAPFILEHEADER bitmapfileheader; // this contains the bitmapfile header
    BITMAPINFOHEADER bitmapinfoheader; // this is all the infoincluding the
palette
    PALETTEENTRY palette[256]; // we will store the palette here
    UCHAR *buffer; // this is a pointer to the data
} // BITMAP_FILE, *BITMAP_FILE_PTR
```

注意我一般将 BITMAPINFOHEADER 和 BITMAPINFO 放在一个结构中，这非常简单。下面是 Load_Bitmap_File()函数。

```
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap, char *filename)
{
    // this function opens a bitmap file and loads the data into bitmap

    int file_handle, // the file handle
        index;      // looping index

    UCHAR *temp_buffer = NULL; // used to convert 24 bit images to 16 bit
    OFSTRUCT file_data;        // the file data information

    // open the file if it exists
```

```

if ((file_handle = OpenFile(filename,&file_data,OF_READ)) == -1)
    return(0);

// now load the bitmap file header
_lread(file_handle, &bitmap->bitmapfileheader,sizeof(BITMAPFILEHEADER));

// test if this is a bitmap file
if (bitmap->bitmapfileheader.bfType!=BITMAP_ID)
{
    // close the file
    _lclose(file_handle);

    // return error:
    return(0);
} // end if

// now we know this is a bitmap, so read in all the sections

// first the bitmap infoheader

// now load the bitmap file header
_lread(file_handle, &bitmap->bitmapinfoheader,sizeof(BITMAPINFOHEADER));

// now load the color palette if there is one
if (bitmap->bitmapinfoheader.biBitCount == 8)
{
    _lread(file_handle, &bitmap->palette,
        MAX_COLORS_PALETTE*sizeof(PALETTEENTRY));

    // now set all the flags in the palette correctly
    // and fix the reversed
    // BGR RGBQUAD data format
    for (index=0; index < MAX_COLORS_PALETTE; index++)
    {
        // reverse the red and green fields
        int temp_color = bitmap->palette[index].peRed;
        bitmap->palette[index].peRed = bitmap->palette[index].peBlue;
        bitmap->palette[index].peBlue = temp_color;

        // always set the flags word to this
        bitmap->palette[index].peFlags = PC_NOCOLLAPSE;
    } // end for index

} // end if

// finally the image data itself
_lseek(file_handle,
    -(int)(bitmap->bitmapinfoheader.biSizeImage),SEEK_END);

```

```
// now read in the image

if (bitmap->bitmapinfoheader.biBitCount == 8 ||
    bitmap->bitmapinfoheader.biBitCount == 16 ||
    bitmap->bitmapinfoheader.biBitCount == 24)
{
    // delete the last image if there was one
    if (bitmap->buffer)
        free(bitmap->buffer);

    // allocate the memory for the image
    if (!(bitmap->buffer =
        (UCHAR *)malloc(bitmap->bitmapinfoheader.biSizeImage)))
    {
        // close the file
        _fclose(file_handle);

        // return error
        return(0);
    } // end if

    // now read it in
    _fread(file_handle, bitmap->buffer,
        bitmap->bitmapinfoheader.biSizeImage);

} // end if
else
{
    // serious problem
    return(0);

} // end else

// close the file
_fclose(file_handle);

// flip the bitmap
Flip_Bitmap(bitmap->buffer,
    bitmap->bitmapinfoheader.biWidth*
    (bitmap->bitmapinfoheader.biBitCount/8),
    bitmap->bitmapinfoheader.biHeight);

// return success
return(1);

} // end Load_Bitmap_File
```

注 意

文件结尾有一个 Filp_Bitmap() 函数，主要是因为大多数 BMP 文件是颠倒格式。Filp_Bitmap() 是我将建立的库的一部分。它被拷贝在下面将要看到的演示程序中，你可以在任何时候阅读它。

函数实际上没有那么长，也没有那么复杂，只是写出来很痛苦。它打开位图文件，调入头信息，装载图形及调色板（若是 256 色）。函数对 8、16、24 位图形适用。但是，不考虑图像格式，存储缓冲 UCHAR 实际上是一个指针。所以，如果图像是 16 位或者 24 位，你必须进行投射或者指针运算。另外，函数给图像分配了一个缓冲。所以缓冲在使用后必须被释放。这通过函数 Unload_Bitmap_file() 来完成。如下所示：

```
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap)
{
    // this function releases all memory associated with the bitmap
    if (bitmap->buffer)
    {
        // release memory
        free(bitmap->buffer);
        // reset pointer
        bitmap->buffer = NULL;
    } // end if

    // return success
    return(1);
} // end Unload_Bitmap_File
```

稍后，我将教你如何将位图文件装载到内存中并显示它们。但是，首先我想讲讲在游戏中是如何利用位图的。

使用位图

多数游戏具有许多艺术作品，包括 2D 精灵、2D 文字、3D 模型等。多数时候，2D 图形以单个图片（见图 7.25）的形式或者是一定数量的图形共同组成的模板（见图 7.26）的形式，一次调入画面。两种方法各有优缺点。调入单个图形，每文件一个图形，最酷的事情是可以用图形处理软件处理它，马上见效。但是，在动画中通常有几百幅图片组成二维角色，这就意味着将出现成百上千个独立的.bmp 图片文件。

模板图像如图 7.26 所示，很伟大，因为它存放这单一动画角色的各个动画画面，因此所有的数据也在同一个文件中。惟一的缺点是必须对数据建模。这可能会花费很多时间，还没有包括对齐的问题，因为你必须创建一个单元模板，每个单元是 $m \times n$ （通常 m, n 是 2 的次幂），在单元周围有一个像素组成的边界。下面，因为你现在知道了单元的大小等，就可以写一软件从特定单元里提取图像。你可能两种技术都用，这要根据游戏的类型，以

及多少个艺术作品在里面而定。许多时候，你不得不写一个程序，从单一的图像文件或者从模板格式文件中提取图像数据，然后将图像发送给 DirectDraw 画面。这允许你使用图形变换，但不妨以后再这样做。现在，仅仅使用 Load_Bitmap_File()函数装载 8、16 位图像，在主缓冲里面显示它们，感受一下函数的作用。

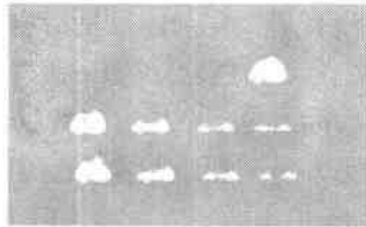


图 7.25 无模板标准系列位图

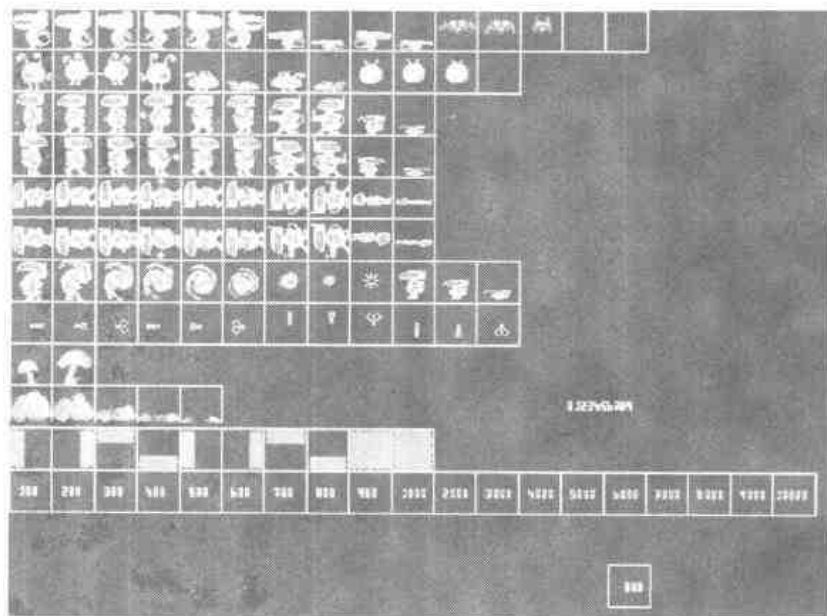


图 7.26 便于使用和扩展的位图图像模板

提示

多数演示图片都是由我自己完成，采用不同的画笔、三维程序，画出的作品存成.bmp 文件(其他的作品是有一些艺术家完成)。一个很好的二维画笔程序，可以支持许多格式。我常常使用的是：Paint Shop Pro，这是一个绝对物有所值的程序，它也在 CD 中。

装载 8 位图像

要装载一个 8 位的图像，应当：

```
BITMAP_FILE bitmap; // this will hold the bitmap
if (!Load_Bitmap_File(&bitmap, "path:\filename.bmp"))
{ /* error */ }
```



```
// do what you will with the data stored in bitmap.buffer
// in addition, the palette is stored in bitmap.palette
// when done, you must release the buffer holding the bitmap
Unload_Bitmap(&bitmap);
```

装载 8 位图像惟一有意思的是在 `BITMAP_FILE` 中的调色板信息有效。你可以使用此数据改变 DirectDraw 的调色板，保存它，等等。这使我们更加清楚写 8 位游戏的细节。

警告



屏幕只能设一次，屏幕 256 色就只能是 256 色，所以当你进行你的艺术品绘制时，记住在 256 色下，看是不是所有的艺术品都能表现很好（Deabelizer 是一个很好的工具）。很多时候，你需要多个调色板——每一关游戏一个——但是，不管怎样，所有的图像将在同一水平下使用，在同一调色板下使用，必须都可见。

现在我们来看一点好东西——在调色板同 8 位 DirectDraw 画面连接之后，改变调色板。如你所知，我们编写的演示程序大多是 8 位模式，创建一个随机的或者梯度的图形，仅仅是那样。但是现在，你将装载具有自己调色板的图像，并希望用一个新的调色板调整 DirectDraw 的调色板，当你将图形拷贝到主缓冲时，才看起来对劲。要做到这点，你只需使用 `IDIRECTDRAWPALETTE:SetEntries()` 函数，如下所示：

```
BITMAP_FILE bitmap; // holds the 8-bit image
// given that the 8-bit image has been loaded
if (FAILED(lpddpal->SetEntries(0, 0, MAX_COLORS_PALETTE,
bitmap.palette)))
{ /* error */ }
```

就是这样简单。

作为 8 位图像装载的例子，请参见演示程序 `DEMO7_10.CPP`。它在 640×480 模式下装入 8 位图像，存在主缓冲中。

装载 16 位位图

装载 16 位位图几乎同装载 8 位位图一样。但是，不用管调色板，因为它没有调色板。很少有画笔程序产生 16 位位图文件，如果想使用 16 位 DirectDraw 模式，你可以装载 24 位位图，然后通过计算转换成 16 位模式。一般，需要用以下操作把 24 位图像转变为 16 位图像。

1. 创建一个 $m \times n$ 字缓冲，每字 16 位。它存放你的 16 位图像。
2. 把 24 位图像装载到你的 `BITMAP_FILE` 结构中。利用你建立的缓冲，用下面的色彩算法把 24 位色彩转变成 16 位。

```
// each pixel in BITMAP_FILE.buffer[] is encoded as 3-bytes
// in BGR order, or BLUE, GREEN, RED
// assuming index is pointing to the next pixel
UCHAR blue = (bitmap.buffer[index*3-0])>>3,
```

```

green = (bitmap.buffer[index*3+1])>>3,
red = (bitmap.buffer[index*3+2])>>3;
// build up 16 bit color word
USHORT color = _RGB16BIT565(red, green, blue);

```

然后，将色彩写入你的 16 位缓冲中。在本书最后，你可以看到所有的库函数。我肯定那里有 24 位向 16 位位图转变的程序。

无论怎样，假设位图实际上是 16 位模式，无需转换，位图装载就应同 8 位的一样。例如，演示程序 DEMO7_11.CPPIEXE 装载 24 位位图，转变成 16 位位图，装入主缓冲。

装载 24 位位图

装载 24 位位图是最简单的。创建一个 24 位位图文件，用 Load_Bitmap_File() 函数装载即可。BITMAP_FILE.buffer[] 将存放它的数据，3 字节从左至右逐行存放，BGR（蓝色，绿色，红色）格式。记住这点，因为它同你提取数据有关。另外，许多图像卡不支持 24 位图像。因为它们不喜欢奇数（3 的倍数）地址，它们支持 32 位。所以作为填充，附加一个字节，或者 Alpha 通道。两种情况下，当你从 BITMAP_FILE.buffer[] 中读出每个像素，写入 32 位 DirectDraw 画面的主画面时，都不得不自己进行填充。下面是一个例子：

```

// each pixel in BITMAP_FILE.buffer[] is encoded as 3-bytes
// in BGR order, or BLUE, GREEN, RED
// assuming index is pointing to the next pixel
UCHAR blue = (bitmap.buffer[index*3+0]),
green = (bitmap.buffer[index*3+1]),
red = (bitmap.buffer[index*3+2]);
// this builds a 32 bit color value in 1.8.8.8 format(8-bit alpha mode)
_RGB32BIT(red, green, blue);

```

你已经看到过这个宏，别走开，下面再次刷新你的记忆：

```

// this builds a 32 bit color value in 1.8.8.8 format(8-bit alpha mode)
#define _RGB32BIT(a, r, g, b) (((g)<<8)+((r)<<8+((b)<<24)))

```

装载 24 位图像的例子请看 DEMO7_12.CPPIEXE，它装载完整的 24 位图像，将显示模式设为 32 位，并将图像拷贝到主画面。

关于位图的后话

好了，可以装载 8、16、24 位格式了。但是，你发现你不得不在下面写很多有用的函数，我来完成它，另外，你可能想装载 TGA 文件，因为许多 3D 模型可以着色的动画序列文件就是以 tga 结尾的 filenamennnn.tga 文件，nnnn 从 0000 到 9999。你可能想需要这样的动画顺序，所以，我在给你讲解库函数时，将给你一个装载 tga 的函数。要比 bmp 简单得多。

备用画面

DirectDraw 的优点就在于它能够利用硬件加速。除非你用 DirectDraw 数据结构和物体存放位图，否则你是不能做到这点的。DirectDraw 是使用图形变换的关键。你已经看到了如何利用建立主画面和后备缓冲创建一个页交换动画链，但是，你仍然需要学习如何在系统缓冲或者 VRAM 中创建一个 $m \times n$ 备用画面。用这些画面，你才能够用位图填充它们，而后利用图形变换将它们从画面图形变换到屏幕。

创建备用画面

除了下面的几点区别，创建备用画面同创建主缓冲一样。

1. 你必须将 DDSURFACEDESC2.dwFlags 设置为 (DDSD_CAPS|DDSD_WIDTH|DDSD_HEIGHT)。
2. 你必须在 DDSURFACEDESC2.dwWidth 和 DDSURFACEDESC2.dwHeight 中设置所需画面的尺寸。
3. 必须将 DDSURFACEDESC2.ddsCaps.dwCaps 设置为 DDSCAPS_OFFSCREENPLAIN | memory_flags，下面 memory_flags 是你想创建画面的位置。如果我将它设置为 DDSCAPS_VIDEOMEMORY，则画面创建在 VRAM 中（如果有空间的话）。如果将它设置为 DDSCAPS_SYSTEMMEMORY，将建立在系统内存中。这使图形变换几乎没用，因为数据需要通过系统总线传输。

作为一个例子，下面的函数给出了你想创建的任何类型的画面。

```
LPDIRECTDRAW_SURFACE4 DDraw_Create_Surface(int width, int height,
                                           int mem_flags)
{
    // this function creates an offscreen plain surface

    DDSURFACEDESC2 ddsd;           // working description
    LPDIRECTDRAW_SURFACE4 lpdds;    // temporary surface

    // initialize structure
    DDRAW_INIT_STRUCT(ddsd);

    // set to access caps, width, and height
    ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;

    // set dimensions of the new bitmap surface
    ddsd.dwWidth = width;
    ddsd.dwHeight = height;

    // set surface to offscreen plain
```

```

ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | mem_flags;

// create the surface
if (FAILED(lpdd->CreateSurface(&ddsd, &lpdds, NULL)))
    return(NULL);

// set color key to color 0
DDCOLORKEY color_key; // used to set color key
color_key.dwColorSpaceLowValue = 0;
color_key.dwColorSpaceHighValue = 0;

// now set the color key for source blitting
lpdds->SetColorKey(DDCKEY_SRCBLT, &color_key);

// return surface
return(lpdds);

} // end DDraw_Create_Surface

```

例如，如果你想在 VRAM 中创建一个 64×64 的画面，可以如下调用。

```

// create surface
if (!space_ship->Ddraw_Create_Surface(64, 64, DDSCAPS_VIDEOMEMORY))
{ /* error */ }

```

技巧



当你创建表面存放位图时，只创建你将多次绘制的图像的 VRAM 表面。另外，按照从大到小的顺序创建它们。

现在，你可以做你想利用画面做的任何事了。例如，可以锁定它，以便将一个位图拷贝到它上面。下面是代码：

```

DDSURFACEDESC2 ddsd; // directdraw surface description
// initialize the structure
DDRAW_INIT_STRUCT();
// lock the surface, check for error in RL(real life)
space_ship->Lock(NULL, &ddsd,
DDLOCK_WAIT | DDLOCL_SURFACEMEMORYPTR,
NULL);};
// do what you will to ddsd.lpSurface and ddsd.lPitch
// unlock
space_ship->Unlock(NULL);

```

用过画面后（游戏结束时），必须用函数 `Release()` 将它们释放给 `DirectDraw`：

```

if (space_ship)
space_ship->Release();

```

这就是利用 DirectDraw 创建备用画面的全部过程。现在，让我们看看如何图形变换到另外一个画面，如后备缓冲或者主画面。

转换备用画面

现在既然知道了装载位图，创建画面，使用图形变换，就可以将它们组合到一起，做一些真正的动画了。这一部分的目标是：装载一些含有物体（船、动物等等）动画画面的位图，创建一些小的画面放置每个动画框架，将位图装入每个画面。一旦所有的画面装载了位图数据，你就要把画面图形变换到屏幕上，让物体动起来。

实际上，你已经知道了完成这些的所有步骤。惟一你没有做过的是用图形变换器从画面而不是后备缓冲刷向主缓冲，但是没有什么不同。如图 7.27 所示，有一些小的画面，每一个有一个不同的动画画面。另外，你还看到一个后备缓冲画面和一个主画面。计划是：将所有位图装入小的画面，用图形变换器将小画面图形变换到后备缓冲上进行页交换看效果。定期将图形变换不同的图像，移动图形变换目的地，使物体移动。

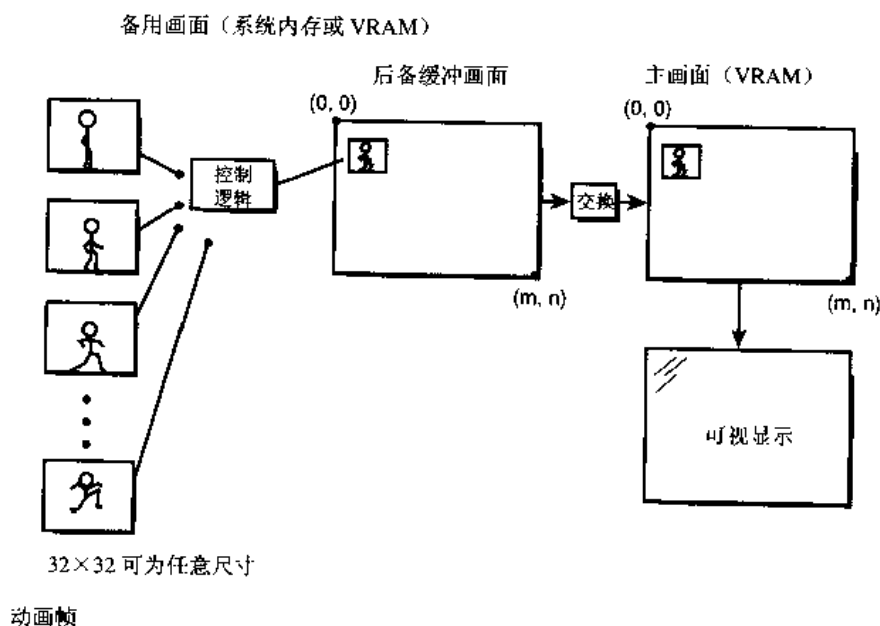


图 7.27 将备用画面图形变换到后备缓冲

设置图形变换器

设置图形变换，需要完成下面工作：

1. 设置要图形变换的源矩形。这是包含有趣图像的小的画面（ 8×8 ， 16×16 ， 64×64 等等）。通常坐标是从 $(0, 0)$ 到 $(\text{宽度}-1, \text{高度}-1)$ ，也就是说整个画面。
2. 设置目标矩形，通常是后备缓冲。这部分有点技巧，因为你想在一定位置 (x, y)

拷贝源图像，所以矩形应该设为从 (x, y) 到 $(x + \text{宽度} - 1, y + \text{高度} - 1)$

3. 用适当的参数调用 `IDIRECTDRAWSURFACE4::Blt()`，稍后会看到。

注 意

如果目标矩形比源矩形大或者小，图形转换将自动缩放使图像合适——这是 2.5D 精灵缩放游戏的基础。

在调用函数 `Blt()` 之前，有一个问题我必须提醒你——色彩关键字。

色彩关键字

色彩关键字较难解释，可能是因为它们的名字在 `DirectDraw` 下变化的缘故。让我来大致说一下。当你进行游戏操作时，多数时候你图形变换的位图物体在一个方框中，但是，当你画一个小动物位图时，你通常不想拷贝方框内的所有内容。你可能想只拷贝同动物有关的位，所以你需要选择一种（或一些）色彩作为透明色。图 7.28 给出了透明和不透明的对比。我在前面已经讨论过这点，你也进行过软件的练习。

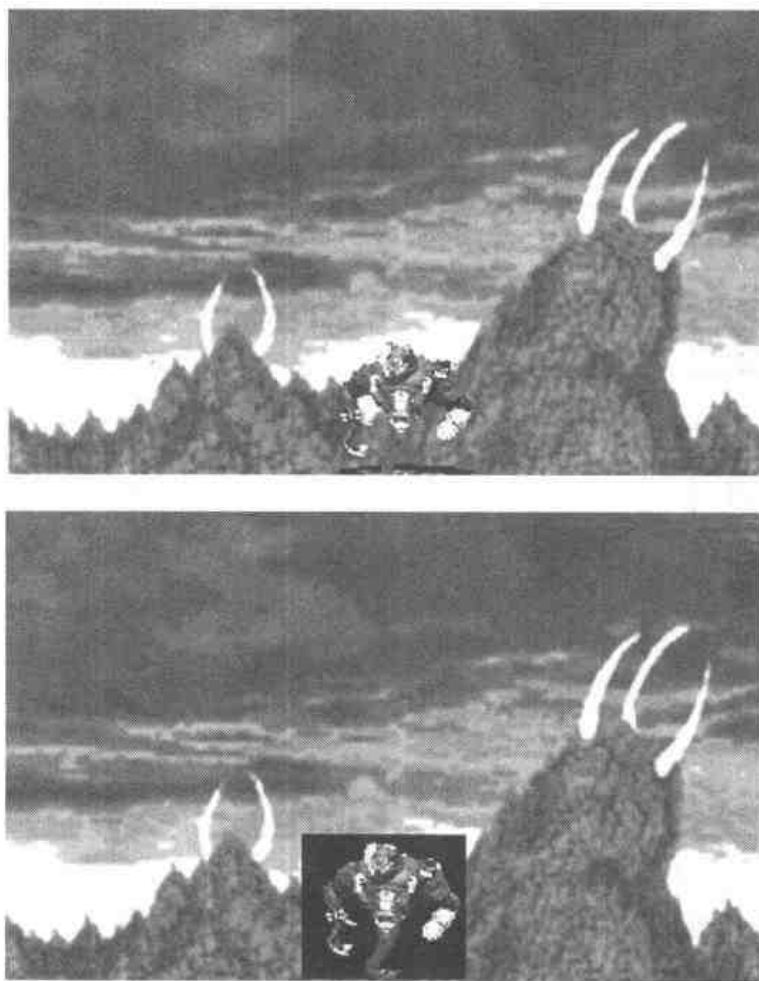


图 7.28 透明图形变换（上）和不透明图形变换（下）的比较

DirectDraw 有一个比选择单一色彩更加精密复杂的色彩关键字系统，它可以完成比采用透明图形变换多得多的事情。让我们快速浏览一下色彩关键字的类型。然后我将给你介绍如何为你感兴趣的操作设置色彩关键字。

源色彩关键字

源色彩关键字是你想用的也是最容易理解的一种。一般而言，你可以选择单一色彩索引（256）或者一定范围的色彩（RGB）作为源图像的透明色。然后，将源刷向目标时，不拷贝具有透明色的像素，图 7.14 给出了这个过程。可以在创建画面时，设置色彩关键字，或者在之后用函数 `IDIRECTDRAWSURFACE4::SetColorKey()` 设置。我将在稍后给你演示两种方法。但是我们首先看一下下面存放色彩关键字的数据结构 `DDCOLORKEY`：

```
typedef struct _DDCOLORKEY
{
    DWORD dwColorSpaceLowValue; // low value (inclusive)
    DWORD dwColorSpaceHighValue; // high value (inclusive)
} DDCOLORKEY, FAR* LPDDCOLORKEY;
```

低位和高位色彩关键字的值有一点技巧。听好了，如果你所用的是 8 位画面，值应该是索引值。如果是 16、24、32 位画面，你实际上采用的是特殊画面格式的 RGB 编码的字来作为低位和高位色彩关键字的值。例如，让我们看看，如果你工作在 8 位色彩下，你想用色彩 0 为透明色，应该如下设置色彩关键字。

```
DDCOLORKEY Key;
key.dwColorSpaceLowValue = 0;
key.dwColorSpaceHighValue = 0;
```

如果你想将 10~20 的色彩作为透明，则：

```
key.dwColorSpaceLowValue = 10;
key.dwColorSpaceHighValue = 20;
```

下面，让我们看看在 16 位的 5.6.5 模式下，如何将纯蓝色作为透明色：

```
key.dwColorSpaceLowValue = _RGB15BIT565(0, 0, 32);
key.dwColorSpaceHighValue = _RGB15BIT565(0, 0, 32);
```

同样，我们看看 16 位模式下，如何将黑色变为半红色以及透明色：

```
key.dwColorSpaceLowValue = _RGB15BIT565(0, 0, 0);
key.dwColorSpaceHighValue = _RGB15BIT565(16, 0, 0);
```

知道了吗？让我们看看在创建过程中，如何设置 DirectDraw 画面色彩关键字。需要你做的是加入一个 `DDSD_CKSRCBLT` 标志（其他可用标志见表 7.5）到描述画面的 `dwFlags` 字中。

然后将低-高色彩关键字分配给 `DDSURFACEDESC2.DdckCKSrcBlit`、`member.dwColorSpaceLowValue` 和 `DDSURFACEDESC2.ddckCKSrcBlit.dwColorSpaceHighValue` 字段（下面也有目标成员和叠加色彩关键字的信息）。

表 7.5 色彩关键字画面标志

值	描 述
<code>DDSD_CKSRCLBLT</code>	表示 <code>DDSURFACEDESC2</code> 的 <code>ddckCKSrcBlit</code> 成员有效，并且包含源色彩关键字的色彩关键字信息
<code>DDSD_CKDESTBLT</code>	表示 <code>DDSURFACEDESC2</code> 的 <code>ddckCKDescBlit</code> 成员有效，并且包含目标色彩关键字的色彩关键字信息
<code>DDSD_CKDESTOVERLAY</code>	表示 <code>DDSURFACEDESC2</code> 的 <code>ddckCKDescOverlay</code> 成员有效，并且包含目标叠加色彩关键字的色彩关键字信息
<code>DDSD_CKSRCLBLT</code>	表示 <code>DDSURFACEDESC2</code> 的 <code>ddckCKSrcBlit</code> 成员有效，并且包含目标色彩关键字的色彩关键字信息
<code>DDSD_CKSRCOVERLAY</code>	表示 <code>DDSURFACEDESC2</code> 的 <code>ddckCKSrcOverlay</code> 成员有效，并且包含源叠加色彩关键字的色彩关键字信息

下面是一个例子：

```

DDSURFACEDESC2 ddsd;           // working description
LPDIRECTDRAW SURFACE4 lpdds;   // temporary surface
// initialize structure
DDRAW_INIT_STRUCT(dds);
// set to access caps, width, and height
dds.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT | DDSD_CKSRCLBLT;
// set dimensions of the new bitmap surface
dds.dwWidth = width;
dds.dwHeight = height;
// set surface to offscreen plain
dds.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN ; mem_flags;
// set the color key fields
dds.ddckCKSrcBlit.dwColorSpaceLowValue = low_color;
dds.ddckCKSrcBlit.dwColorSpaceHighValue = high_color;
// create the surface
if (FAILED(lpdd->CreateSurface (&dds, &lpdds, NULL)))
return(NULL);

```

一旦创建了一个带有或没有色彩关键字的画面，你可以用函数 `IDIRECTDRAW SURFACE4:SetColorKey()` 来设置它：

```

HRESULT SetColorKey(DWORD dwflags,
LPDDCOLORKEY lpDDColorKey);

```


有效标志列于表 7.6。

表 7.6 SetColorKey() 的有效标志

值	描 述
DDCKEY_COLORSPACE	表示该结构包含一个色彩空格。在设定一个色彩范围时必须设定该位
DDCKEY_SRCBLT	表示该结构指定一个色彩关键字或色彩空格作为图形变换操作的源色彩关键字
DDCKEY_DESTBLT	表示该结构指定一个色彩关键字或色彩空格作为图形变换操作的目标色彩关键字
DDCKEY_DESTOVERLAY	如果该结构指定一个色彩关键字或色彩空格作为叠加操作的目标色彩关键字，则设定该标志（高级）
DDCKEY_SRCOVERLAY	如果该结构指定一个色彩关键字或色彩空格作为叠加操作的源色彩关键字，则设定该标志（高级）

下面给出一个例子：

```
// assume lpdds points to a valid surface
// set color key
DDCOLORKEY color_key; // used to set color key
color_key.dwColorSpaceLowValue = low_value;
color_key.dwColorSpaceHighValue = high_value;

// now set the color key for source blitting, notice
// the use of DDCKEY_SRCBLT
lpdds->SetColorKey(DDCKEY_SRCBLT, &color_key);
```

提 示



如果你为源关键字设置了一个色彩范围，在调用 SetColorKey() 时，必须加上标志 DDCKEY_COLORSPACE，如：

```
lpdds->SetColorKey(DDCKEY_SRCBLT | DDCKEY_COLORSPACE,
&color_key);
```

否则，DirectDraw 将使色彩范围退化为一个值。

目标色彩关键字

理论上目标色彩关键字作用很大，但是从没有见过使用，目标色彩关键字的概念如图 7.29 所示。基本思想是：你可以在目标画面上设一种或者一个色彩范围，不能够被图形变换。本质上，你是在创建一种掩码。这个方法可以模拟窗口或护栏等。

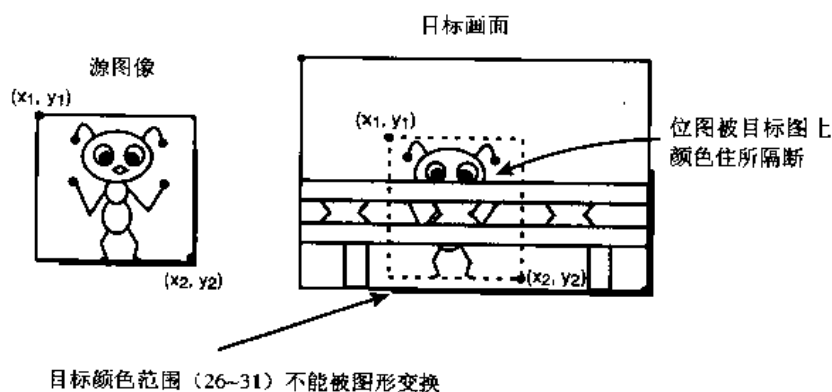


图 7.29 目标色彩关键字

设置目标色彩关键字的方法与源的相同，只需要改变两个标志。例如，在创建画面设 `DDRAWSURFACEDESC2.dwFlags` 时，目标色彩关键字的设置需要将 `DDSD_CKSRCLBLT` 换成 `DDSD_CKDESTBLT`，当然，还要将关键字的值设在 `ddsd.ddckCKDestBlt` 中，而不是设在 `ddsd.ddckCKSrcBlt` 中。

```
// set the color key fields
ddsd.ddclCKDestBlt.dwColorSpaceLowValue = low_value;
ddsd.ddclCKDestBlt.dwColorSpaceHighValue = high_value;
```

如果你想在创建画面之后创建目标色彩关键字，除了调用函数 `SetColorKey()` 使 `DDCKEY_SRCBLT` 标志换成 `DDCKEY_DESTBLT` 之外，一切都一样。如下所示：

```
Lpdds->SetColorKey(DDCKEY_DESTBLT, &color_key);
```

警告



目标色彩关键字现在只是在 HAL（硬件抽象层）可用，在 HEL 中则不行，因此，如果没有硬件支持目标色彩关键字，它就不起作用。在将来版本的 DirectX 中可能实现这一点。

最后，还有另外两种色彩关键字：源覆盖图和目标覆盖图。它们对你没有用，是为视频处理设计的。如果有兴趣，可参看 DirectX SDK。

使用图形变换

现在，你已经有足够的预备知识了，将一个备用画面变换到其他画面不过是小菜一碟。下面就是：假设创建了一个 64×64 像素，8 位色彩的画面图像，具有 0 索引透明色，或者其他一些像下面这样的东西：

```

DDSURFACEDESC2 ddsd; // working description
LPDIRECTDRAWSURFACE4 lpdds_image; // temporary surface

// initialize structure
DDRAW_INIT_STRUCT(dds);
// set to access caps, width, and height
dds.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT | DDSD_CKSRCBLT;
// set dimensions of the new bitmap surface
dds.dwWidth = 64;
dds.dwHeight = 64;
// set surface to offscreen plain
dds.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | mem_flags;
// set the color key fields
dds.ddckCKSrcBlt.dwColorSpaceLowValue = 0;
dds.ddckCKSrcBlt.dwColorSpaceHighValue = 0;
// create the surface
if (FAILED(lpdd->CreateSurface(&dds, &lpdds_image, NULL));
return(NULL);

```

下面,假设你有主画面 `lpddsprimary`,后备缓冲画面 `lpddsback`,并且想把画面 `lpdds_image` 用源色彩关键字图形变换到后备缓冲画面的 (x, y) 处,可以这样做:

```

// fill in the destination rect
dest_rect.left = x;
dest_rect.top = y;
dest_rect.right = x+64-1;
dest_rect.bottom = y+64-1;

// fill in the source rect
source_rect.left = 0;
source_rect.top = 0;
source_rect.right = 64-1;
source_rect.bottom = 64-1;

// blt to destination surface
if (FAILED(lpddsback->Blt(&dest_rect, lpdds_image,
&source_rect,
(DDBLT_WAIT | DDBLT_KEYSRC),
NULL)))
Return(0);

```

就是这样。注意标志 `DDBLT_KEYSRC`,你在图形变换调用时,必须有它。否则,即使你在画面上定义了色彩关键字,它也不起作用。

警告



当你进行图形变换时,对剪切要特别小心。不要将剪切设到目标表面,在它上面图形变换会很糟糕。你需要做的就是调用你的 `Ddraw_Attach_Clipper()` 函数,设一个同屏幕边界一样的单独的剪切矩形。

最后，当然就可以做一个很酷的演示程序了，图 7.30 是演示程序 DEMO7_13.CPPIEXE 的一个画面。酷吧？我想给这个游戏进一步编程，使你能够从这个演示程序中获得比一些移动图像更多的东西。一般，演示程序都是装载一个很大的位图作背景，一些动画框作为外来物（一些画面）。外来物以不同的速度移动或动作。看看你能不能够加入一受到箭头键控制的玩家到演示程序中！

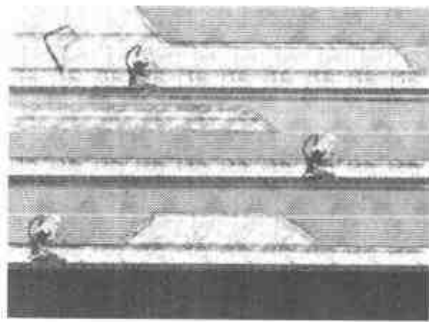


图 7.30 运行 DEMO7_13.EXE 文件

位图的旋转和缩放

如图 7.31 所示，DirectDraw 支持两种位图的旋转和缩放。但是，只有 HAL 支持旋转。这就意味着如果没有硬件支持旋转，将很不幸。你也许会问，“为什么 HEL 仅仅支持缩放，而不支持旋转？”。答案是因为位图的旋转比缩放要慢 10~100 倍。微软发现，无论它们如何修改旋转代码，还是太慢。所以，你在任何时候都可以考虑采用缩放，而不是旋转。你可以写你自己的旋转程序，但是相当复杂，对 3D 多边形游戏来说，也没有必要。本书中不包括这方面的内容。

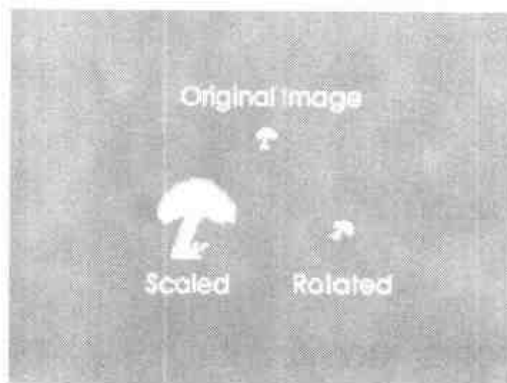


图 7.31 位图缩放和旋转

执行位图的缩放很简单。需要你做的就是改变目标矩阵的尺寸，同源矩阵的尺寸不同，图像就会被缩放。例如，假设有一个 64×64 的图像，你想将它缩放到 $m \times n$ 的尺寸，位于 (x, y) 处。代码如下：

```

// fill in the destination rect
dest_rect.left  = x;
dest_rect.top   = y;
dest_rect.right = x+m-1;
dest_rect.bottom= y+n-1;

// fill in the source rect
source_rect.left  = 0;
source_rect.top   = 0;
source_rect.right = 64-1;
source_rect.bottom= 64-1;

// blt to destination surface
if (FAILED(lpddsback->Blt(&dest_rect, lpdds_image,
&source_rect,
(DDBLT_WAIT | DDBLT_KEYSRC),
NULL)))
Return(0);

```

很简单。旋转有点难，因为你不得不设一个 **DDBLTFX** 结构。执行旋转操作，你必须要有加速硬件支持它（很贵），然后设置 **DDBLTFX** 结构。如下所示：

```

DDBLTFX ddbltfx; // this holds our data
// initialize structure
DDRAW_INIT_STRUCT(ddbltfx);
// set rotation angle, note that each unit is in 1/100
// of a degree rotation
ddbltfx.dwRotationAngle = angle; // each unit is

```

然后，像以往那样调用 **Blt()**，但是，需要加上 **DDBLT_ROTATIONANGLE** 到标志参数，并像下面这样加上 **ddbltfx** 参数：

```

// blt to destination surface
if (FAILED(lpddsback->Blt(&dest_rect, lpdds_image,
&source_rect,
(DDBLT_WAIT | DDBLT_KEYSRC | DDBLT_ROTATIONANGLE),
&ddbltfx)))
Return(0);

```

警告



你可以通过查询一个表面的 **DDSCAPS** 结构性能和察看 **DDSCAPS** 的成员 **dwFxCaps** 的 **DDFXCAPS_BLTROTATION*caps** 标志来确定你的硬件是否支持转动。你可以通过函数 **IDIRECTDRAWSURFACE4::Get Caps()** 来查询表面的性能。这将在本章末尾介绍。

如果你的硬件支持旋转，位图就旋转了。

在进行 **DirectDraw** 的旋转和缩放演示程序之前，我想至少应当先谈谈以软件的方式的采样理论和如何完成缩放。

离散采样理论

这里只作简介：我想在第二部分的 3D 文字影像再详细讨论该理论，现在只是一个大概。

当运用位图时，实际上是运用符号；这些符号只是些离散的 2D 图像数据，不是像收音机信号那样的模拟连续信号。两种情况都可以使用信号处理技术，更精确一点是图像的数字信号处理技术。我们感兴趣的地方是数据采样和映射。

在 2D 或 3D 图像领域，当你想从一个位图图像采样，然后在其上执行一些操作时，如缩放、旋转、文字映射等，将花费很多时间。有两种类型的映射：向前映射和反向映射。图 7.32 和 7.33 给出了图例。

一般，向前映射是指像素从源向目标映射或沉积，惟一的问题是在映射过程中，在目标上的像素可能没有从源映射到，这取决于映射函数的选择。

换句话说，反向映射更好一些，它用目标中的每一个像素寻找其源像素。当然，这也有一个问题，一些目标上的像素不得被复制，因为源没有足够的数据填充目标。这个问题将导致图形失真。当有太多数据时，也会产生图形失真，但是可以用平均或者其他数学过滤器将之变小。也就是说，数据多总比数据少好。

两种映射操作都可用于缩放，但是，我们将采用反向映射。让我来给你演示如何缩放一维位图，从它的算法，你就可以生成二维缩放的算法。这点很重要：许多算法是可独立的，意味着图像在多维情况下，可以采用类似的方法处理。一个轴的处理结果不影响另外一个轴。

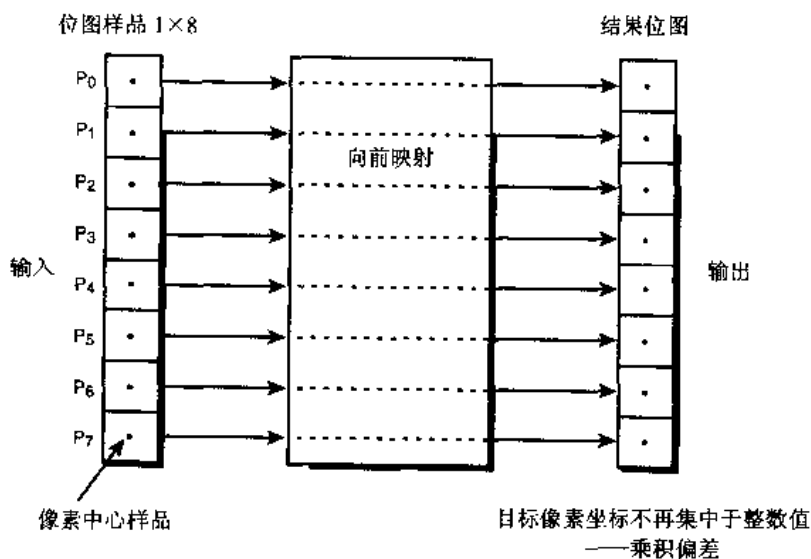


图 7.32 采样原理：向前映射

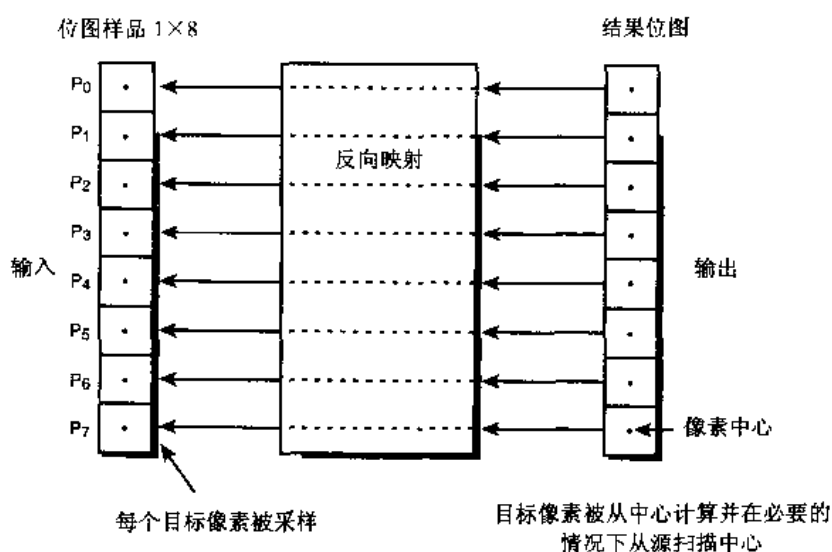


图 7.33 采样原理：反向映射

- 实例 1：假设你有一个 1×4 像素位图，想将它缩放为 1×8 位图。图 7.34 给出了结果。基本上，是将源的每一点像素向目标拷贝了两遍。

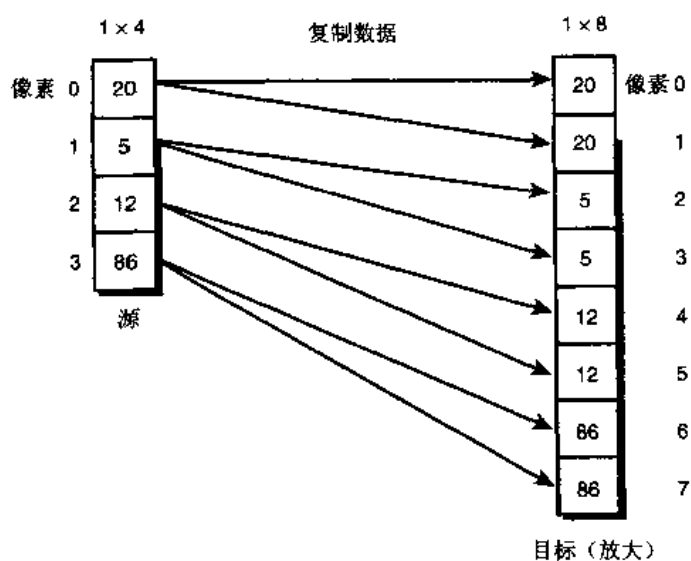


图 7.34 将 1×4 像素位图放大为 1×8 像素

实例 2：假设你有一个 1×4 像素位图，想将它缩放为 1×2 位图。图 7.35 给出了结果。基本上，是将源的像素丢弃了两个。这就产生一个问题：你丢掉了信息。它还正确吗？从数据被丢失的角度看不正确。但是从它所起作用且速度很快的角度来说正确。

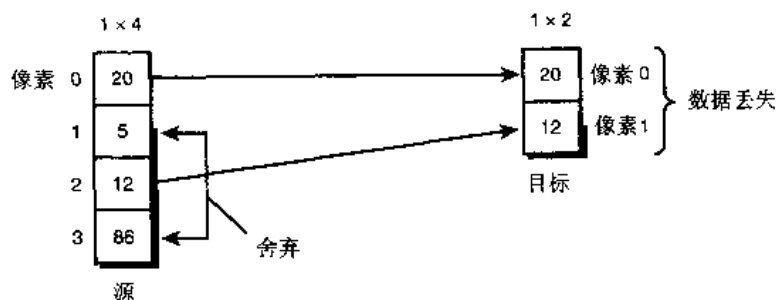


图 7.35 将 1×4 像素位图缩小为 1×2 像素

在上述两个例子中，更好的做法是采用过滤器进行处理。例如，在例子 1 中，你只是对像素进行了拷贝，但是还可以采用两个相邻像素的平均值作为额外的像素点。这会使得缩放看起来更好。这就是两点线性图形过滤器的由来。它是基于这个原理，但只适合在二维情况下。在例子 2 中，你可以使用一个过滤器做同样的事情，对两个像素进行平均，尽管是要扔掉两个点，但可以将它们的信息保存在剩余的像素中，会使结果看起来更自然。

我将在本书最后将给你演示过滤器的应用，所以，现在只是用原始的手段进行缩放。看这个例子，你可以注意到我们通过一定比率（采样比率）对源进行采样，并且基于采样比率填充目标。数学上可以这样完成：

```
// the source height of the 1D bitmap
float source_height;
// the destination height of the desired scaled 1D bitmap
float dest_height;
// the sample rate
float sample_rate = source_height/destination_height;
// used to index source data
float sample_index = 0;
// generate scaled destination bitmap
for (index = 0; index < dest_height; index++)
{
    // write pixel
    dest_bitmap[index] =
    source_bitmap[(int)sample_index];
    // advanced source index
    sample_index+=sample_rate;
} // end for index
```

这就是缩放位图所需要的全部代码。当然，还应当加上其他尺寸。我在数学上舍弃了浮点运算，这是速度的需要。

假设一个源位图是 1×4，像下面这样：


```

source_bitmap[0] = 12
source_bitmap[1] = 34
source_bitmap[2] = 56
source_bitmap[3] = 90

```

现在，将它放大为 1×8 ：

```

Set source_height = 4
dest_height       = 8
sample_rate       = 4/8=0.5

```

运行算法(四舍五入)

index	sample_index	dest_bitmap[index]
0	0	12
1	0.5	12
2	1.0	34
3	1.5	34
4	2.0	56
5	2.5	56
6	3.0	90
7	3.5	90

不错!每个像素正好复制了两次，现在，试试更复杂的，将位图缩放到高度 3 像素。

```

Set source_height = 4
dest_height       = 3
sample_rate       = 4/3=1.333

```

运行算法(四舍五入)

index	sample_index	dest_bitmap[index]
0	0	12
1	1.333	34
2	2.666	56

注意到你在源中丢失了最后一点像素——90。不知你是否喜欢这样。也许你想在缩放 1×2 或者更大的位图时，想看到顶/底部像素，而舍弃中间的像素。这就是四舍五入和偏离 Sample_rate 和 Sample_index 的用武之地——想一想……

现在，既然知道了如何缩放，让 DirectDraw 给你实施。DEMO7_14.CPP1EXE 是 DEMO7_13.CPP1EXE 的翻版。但是我在其中加入了武断的缩放，使它们看起来具有不同的尺寸。如果你有硬件支持缩放，演示会非常顺利。但是如果你没有，可能会有一些停顿。你将在本章后面部分，再一次看到如何使用函数 IDirectDrawSurface4::GetCaps()检测。

色彩效果

我想讨论的下一个主题，是关于色彩动画和诀窍。过去，256 色调色板是惟一可用的位深度，人们发明了许多利用色彩改变现象的技术，也就是，在屏幕上，改变一个或多个调色板寄存器是瞬间可见的。现在，256 色彩方式随着更快的硬件或者加速硬件的普及会渐渐退出市场，但是学习这些技术对我们理解相关的概念是有用的，况且完全使用 RGB 还有待时日。另外还有许多 486 或者慢一些的奔腾处理器，以及只能够以一定的速度处理 256 色彩模式的机器。

256 色模式色彩动画

色彩动画是指在运行中更改或移动色彩盘，例如，发光物质、闪烁的光线、许多可以通过简单操纵运行中的色彩表，就可以得到的许多特殊色彩效果。最酷的事情是在屏幕上的任何物体，只要可以通过色彩列表操纵其像素值，都会被影响到。

想像一下，通过位图实现这点有多么难。例如你有一个小船，上面有亮着的灯，你想让它闪亮。你就需要每幅动画一个位图。但是，利用色彩动画，你只需要一个位图，给灯一个特定的色彩索引，然后使色彩索引活动即可。图 7.36 直接说明了这点。

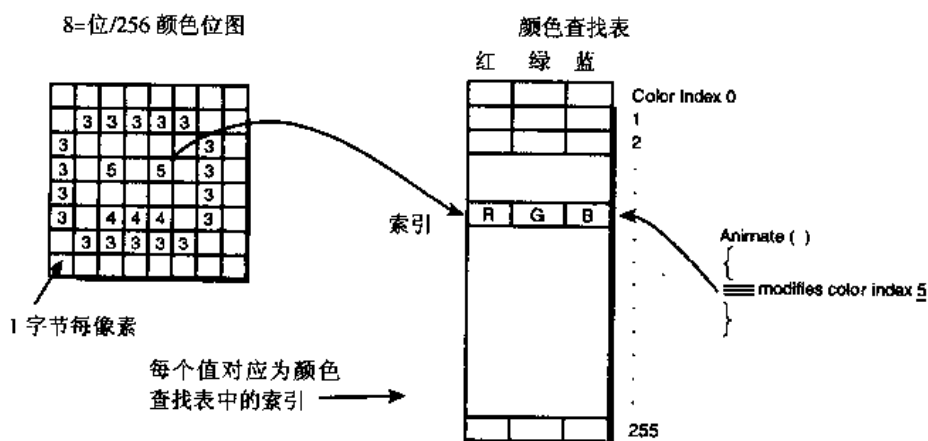


图 7.36 采用间接调色板的色彩动画

我喜欢的两种效果是闪亮和发光。然我们来从闪亮光函数开始。下面是你想得到的功能。

- 创建一束 256 色闪烁光。
- 每束光有一个开关色彩。在开和关之间有一个延时。
- 通过标识符在任意时候开关任意闪烁灯，并且/或者设置其参数。

- 中断光的闪烁恢复其存储数据。

这是一个很好的演示一些稳定数据技术和调节调色板项目的技术。我的战略是创建一个函数，用来创建或者销毁灯，同执行动画一样。函数使用局域静态数组，有几种操作模式。

BLINKER_ADD——用来为基本数据添加闪烁色，当调用时，函数返回用于闪烁的标识符数。系统采用 256 色。

BLINKER_DELETE——删除所发送的标识符的闪烁光。

BLINKER_UPDATE——调整所发送的标识符光的开关参数。

BLINKER_RUN——处理一周期内所有的光。

存储一束光的数据结构是 **BLINKER**，如下所示：

```
// blinking light structure
typedef struct BLINKER_TYP
{
    // user sets these
    int color_index;           // index of color to blink
    PALETTEENTRY on_color;    // RGB value of "on" color
    PALETTEENTRY off_color;   // RGB value of "off" color
    int on_time;               // number of frames to keep "on"
    int off_time;              // number of frames to keep "off"
    // internal member
    int counter;               // counter for state transitions
    int state;                 // state of light
                                // -1 off, 1 on, 0 dead
} BLINKER, *BLINKER_PTR;
```

通常，需要你用 **BLINKER_ADD** 填充“用户”区域，之后调用函数。无论怎样，一般的操作有：在任何时候调用函数进行加、删、调整，但是，一个屏幕只能够用运行命令一次。下面给出函数的代码：

```
int Blink_Colors(int command, BLINKER_PTR new_light, int id)
{
    // this function blinks a set of lights

    static BLINKER lights[256]; // supports up to 256 blinking lights
    static int initialized = 0; // tracks if function has initialized

    // test if this is the first time function has run
    if (!initialized)
    {
        // set initialized
        initialized = 1;

        // clear out all structures
```

```

    memset((void *)lights,0, sizeof(lights));

    } // end if

// now test what command user is sending
switch (command)
{
    case BLINKER_ADD: // add a light to the database
    {
        // run thru database and find an open light
        for (int index=0; index < 256; index++)
        {
            // is this light available?
            if (lights[index].state == 0)
            {
                // set light up
                lights[index] = *new_light;

                // set internal fields up
                lights[index].counter = 0;
                lights[index].state = -1; // off

                // update palette entry
                lpddpal->SetEntries(0,lights[index].color_index,
                                    1,&lights[index].off_color);

                // return id to caller
                return(index);

            } // end if

        } // end for index

    } break;

    case BLINKER_DELETE: // delete the light indicated by id
    {
        // delete the light sent in id
        if (lights[id].state != 0)
        {
            // kill the light
            memset((void *)&lights[id],0,sizeof(BLINKER));

            // return id
            return(id);

        } // end if
    } else
        return(-1); // problem
}

```

```

    } break;

case BLINKER_UPDATE: // update the light indicated by id
{
    // make sure light is active
    if (lights[id].state != 0)
    {
        // update on/off parms only
        lights[id].on_color = new_light->on_color;
        lights[id].off_color = new_light->off_color;
        lights[id].on_time = new_light->on_time;
        lights[id].off_time = new_light->off_time;

        // update palette entry
        if (lights[id].state == -1)
            lpddpal->SetEntries(0,lights[id].color_index,
                                1,&lights[id].off_color);
        else
            lpddpal->SetEntries(0,lights[id].color_index,
                                1,&lights[id].on_color);

        // return id
        return(id);

    } // end if
    else
        return(-1); // problem

    } break;

case BLINKER_RUN: // run the algorithm
{
    // run thru database and process each light
    for (int index=0; index < 256; index++)
    {
        // is this active?
        if (lights[index].state == -1)
        {
            // update counter
            if (++lights[index].counter >= lights[index].off_time)
            {
                // reset counter
                lights[index].counter = 0;

                // change states
                lights[index].state = -lights[index].state;

                // update color

```

```

        lpddpal->SetEntries(0,lights[index].color_index,
                           1,&lights[index].on_color);

    } // end if

    } // end if
else
if (lights[index].state == 1)
{
    // update counter
    if (++lights[index].counter >= lights[index].on_time)
    {
        // reset counter
        lights[index].counter = 0;

        // change states
        lights[index].state = -lights[index].state;
        // update color
        lpddpal->SetEntries(0,lights[index].color_index,
                           1,&lights[index].off_color);

    } // end if
    } // end else if

    } // end for index

    } break;

default: break;

} // end switch

// return success
return(1);

} // end Blink_Colors

```

注 意

我对调整 DirectDraw 调色板项目的部分加黑了，程序中假定有一个全局调色板接口 lpddpal。

函数主要包括三部分，初始化、调整、运行逻辑。当函数第一次调用时，它进行自身初始化。然后是调整代码段或者运行段。如果一个调整类型命令被请求，执行逻辑运算运行加、删、调整闪烁光。如果运行模式被请求，光在一个周期内全部处理。一般，你在首次加入一个或多个灯光时，调用该函数。这时，你应该通过设置 **BLINKER** 结构，然后用

一个 **BLINKER_ADD** 命令，将结构传送给函数。函数然后返回你的闪烁灯的标识符，你应该将它保存，如果想要删除或者调整闪烁灯时，还需要该标识符。

在创建完毕所有想要的灯光后，可将以上所有参数（除了 **BLINKER_RUN**）设为 **NULL** 调用函数。对游戏循环中的每一个画面这么做。例如，假设你有一个游戏，运行在 30fps，你想用一个红色的，占空比 50% 的——一秒开，一秒关的灯和一个绿色的占空比 50% 的——两秒开，两秒关的灯。更进一步说，你想分别使用 250、251 为红色和绿色的调色板索引项目，需要以下代码：

```
BLINKER temp; // used to hold temp info
PALETTEENTRY red = {255, 0, 0, PC_NOCOLLAPSE};
PALETTEENTRY green = {0, 255, 0, PC_NOCOLLAPSE};
PALETTEENTRY black = {0, 0, 0, PC_NOCOLLAPSE};

// add red light
temp.color_index = 250;
temp.on_color = red;
temp.off_color = black;
temp.on_time = 30; // 30 cycles at 30fps = 1sec
temp.off_time = 30;

// make call
int red_id = Blink_Colors(BLINKER_ADD, &temp, 0);

// now create green light
temp.color_index = 251;
temp.on_color = green;
temp.off_color = black;
temp.on_time = 60; // 30 cycles at 30fps = 2sec
temp.off_time = 60;

// make call
int green_id = Blink_Colors(BLINKER_ADD, &temp, 0);
```

现在，准备开始摇滚吧！在你的程序的循环的主要部分，你将每次循环都调用 **Blink_Color()**，如下所示：

```
// enter main event loop
while(TRUE)
{
    // test if there is a message in queue, if so get it
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // test if this is a quit
        if (msg.message == WM_QUIT)
            break;
```

```
// translate any accelerator keys
TranslateMessage(&msg);

// send the message to the window proc.
DispatchMessage(&msg);
} // end if

// main game processing goes here
Game_Main()

// blink all the colors
// could put this into Game_Main() also -better idea
Blinker_Colors(BLINKER_RUN, NULL, 0);

} // end while
```

当然，你可以在任何时候通过其标识符删除闪烁灯。它就不会再次被处理了。例如，如果想去掉红色灯：

```
Blinker_Colors(BLINKER_DELETE, NULL, red_id);
```

就这么简单，当然，你可以通过设置另外一个 **BLINKER** 结构，用 **BLINKER_UPDATE** 调用函数，就可以调整闪烁灯的开关时间和色彩值。例如，如果想改变绿色闪烁灯的参数：

```
// set new parms
temp.on_time    = 100;
temp.off_time   = 200;
temp.on_color   = {255, 255, 0, PC_NOCOLLAPSE};
temp.off_color  = {0, 0, 0, PC_NOCOLLAPSE};

// update blinker
Blink_Colors (BLINKER_UPDATE, temp, green_id);
```

足够了！看 **DEMO7_15.CPP\EXE**，它利用了 **Blink_Colors()** 函数在一个开着的小船上放置了一些闪烁灯。

256 色模式下的色彩旋转

下一个有趣的动画效果是色彩旋转或者色彩移动。通常，它是将相比邻的色彩项目或者寄存器收集在一起，然后以圆周的方式移动它们，如图 7.37 所示。采用这种技术，不需要向屏幕写一个像素，你可以使得物体看起来像是在移动。模拟水流动的效果非常好。另外，你可以在不同位置绘制一些图像，每一个具有不同的色彩索引。然后，如果色彩旋转，就如同物体在移动。**Great 3D Star War** 中的战壕可能就是这样做的。

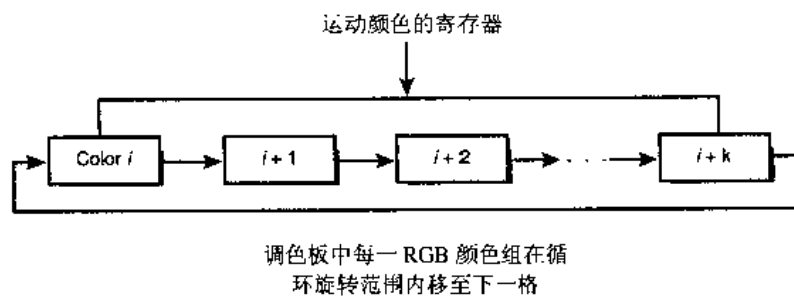


图 7.37 色彩旋转

色彩旋转的代码很简单，算法是采用下面的代码旋转色彩 `color[c1]` 到 `color[c2]`。

```
temp = color[c1]

for (int index = c1; index < c2; index++)
    color[c1] = color[c1+1];
// finish the cycle, close the loop
color[index] = temp;
```

下面是通过我们的库函数完成了算法的一个函数。

```
int Rotate_Colors(int start_index, int end_index)
{
    // this function rotates the color between start and end
    int colors = end_index - start_index + 1
    PALETTEENTRY work_pal[MAX_COLORS_PALETTE]; // working palette
    // get the color palette
    lpddpal->GetEntries(0, start_index, colors, work_pal);
    // shift the color
    lpddpal->SetEntries(0, start_index+1, colors-1, work_pal);
    // fit up the last color
    lpddpal->SetEntries(0, start_index, 1, &work_pal[colors - 1]);
    // update shadow palette
    lpddpal->GetEntries(0, 0, MAX_COLORS_PALETTE, palette);

    // return success
    return(1);
} // end Rotate_Colors
```

通常，算法采用你想旋转的起始点和终止点执行旋转。不要担心“阴影调色板”部分，这是一个函数库，把注意力集中到逻辑部分。有趣的是算法工作原理。它完成的是同 `for` 循环语句一样的功能，但是采用了不同的路子。通过一个适当的移动使其成为可能。下面，看看 DEMO7_16.CPPIEXE，它采用了此函数创建了移动着的酸水。

使用 RGB 模式的技巧

采用 RGB 模式的问题是没有任何色彩间接寻址。换句话说，在屏幕上的每一种像素都有它的色彩值。没有办法制作一个变动可以影响到整个图像。但是，有两个方法可以进行色彩的相关联处理：

- 使用人工色彩转换或者查询表。
- 采用新的 DirectX 色彩和 Gamma 修正子系统，在主画面上实行色彩操纵。

技巧



Gamma 修正处理计算机显示器的非线性反映给输入设备。许多时候，Gamma 修正允许你修改视频信号的强度和亮度。但是，Gamma 修正允许分别改变红色、绿色、蓝色的通道以获得有趣的效果。

人工色彩变换或者查询表

退一万步讲，你还是能够使用 Gamma 修正系统对整个屏幕的图像进行过滤操作。不管怎样，我将首先讨论 RGB 模式下的查询表，然后再讨论 DirectX Gamma 修正系统。

当处理以 RGB 字编码的像素时，实际上没有办法实现色彩动画。你不但必须为改变的色彩写每一像素，还可能不得不读出它。因此，最差的情况下，你可能必须对每一想操纵的像素读、传输、写入周期。没有简单的方法。

但是，还是有一些办法能帮助你，多数时候，在 RGB 空间实行数学转换非常费力。例如，假设你想用一个 16 位图形模式在 (x, y) 处，以一定尺寸（宽，高）模拟一个正方形点光源。怎么做呢？

你先从扫描组成点光源区域的四边形像素开始，并把它存储在一个位图中。然后，对于位图中的每一个像素，执行下面这样的色彩转换：

```
I*pixel(r,g,b) = pixel(I*r, I*g, I*b)
```

为了改变强度就需要三次乘法运算。还不包括你必须首先从 RGB 字中提取 16 位字的分量，在转换完成后还需要把 16 位 RGB 组合在一起。下面的技巧就是采用查询表。

不是采用 16 位模式下所有的 65536 种色彩，你画的物体使用的是要能够发光的色彩，也就是说，均匀分布在 64K 空间的 1024 种色彩。然后，你可以创建一个二维数组查询表，具有 1024 次或者你想要的强度等级，如 64。之后，利用每个实际色彩的 RGB 水平，计算它的 64 色调。把它们的每一个存储在表中。当你创建点光源时，使用 16 位字作为表的索引。同时把光的等级做第二个索引。表中的 16 位结果就是预先建模的 RGB 值。所以，光操作就是一个简单的查询。

这种技术可以被用来透明、Alpha 混合、光照、黑暗化等处理。直到下一章，我才给

出它的演示程序。但是，如果你想在 16、24、32 位色彩模式下使用光照或者色彩效果，还想有一定的速度，查询表是惟一的方法。

新的 DirectX 色彩和 Gamma 控制接口

在 DirectX5.0 中，增加了两个写的接口，帮助游戏程序员和视频程序员获得对屏幕图像的色彩性能更多的控制，无需借助于复杂的算法。例如，对屏幕上的图像添加一点红色、改变色彩等等。但是，像这样的操作在电视机上就像旋转一个钮一样简单，采用数据软件来作这件事是相当复杂的。谢天谢地，有两个新的接口，IDirectDrawGammaControl 和 IDirectDrawColorControl 让程序员通过一些简单的调用就可以做到这些改变。

IDirectDrawColorControl 和电视接口控制非常相似，使你可以控制亮度、对比度、色调、饱和度、锐化和 Gamma 值。要用这个接口，你必须通过 IID_IDirectDrawColorControl 标志从组画面指针中访问它。一旦获得接口，就设置一个 DDColorControl 结构，如下所示：

```
typedef struct _DDCOLORCONTROL
{
    DWORD    dwSize; // size of this struct
    DWORD    dwFlags; // indicates which fields are valid
    LONG     lBrightness;
    LONG     lContrast;
    LONG     lHue;
    LONG     lSaturation;
    LONG     lSharpness;
    LONG     lGamma;
    LONG     lColorEnable;
    DWORD    dwReserved1;
} DDColorControl, FAR *LPDDColorControl;
```

然后，调用 IDIRECTDRAWCOLORCONTROL::SetColorControl()，主画面立即被改动。变化一直保持到第二次调用。

```
HRESULT SetColorControl(LPDDColorControl lpColorControl);
```

Gamma 控制有点不同，不像所有的“电视设置”那样，Gamma 修正控制让你控制主画面的红色、绿色、蓝色扫描。基本上，你定义的扫描的形状决定红色、绿色、蓝色的色彩反映。设置 IDirectDrawGammaControl 同色彩控制相类似，所以不再重复。从 DirectX SDK 中多了解这个主题，因为它可以非常容易地获得许多效果，如水下的感觉、屏幕闪烁、灯光、黑暗等等。惟一的问题是不知道你的硬件是否支持它，很少有硬件支持它，我的也没有一个支持，所以我也没有办法做一个演示程序。

GDI 和 DirectX 混合使用

啊.....对不起, 我只是需要释放一些压力。现在, 回到 GDI, 或者说图形设备接口, 是 WIN32 对应所有窗口着色的图形子系统。在前面 Windows 编程部分, 你已经看到过如何应用 GDI, 所以我不想再回到设备上下文等之类的内容。

在 DirectDraw 下用 GDI, 需要你来做的是在 DirectDraw 中重载一个兼容 DC, 使用它就像是用从标准的 GetDC()调用获得的 DC 一样。最酷的事情是一旦你从 DirectDraw 重载了 GDI 兼容 DC, 在如何使用 GDI 函数上就没有真正的不同。实际上, 你的所有代码不需要改变就可以工作。

现在, 你可能会疑惑, 如果 DirectDraw 接管了图形系统, GDI 怎么能够在 DirectDraw 下就像在全屏幕下一样工作呢?好!在独占模式下, Windows 不能够用 GDI 在你的任何 DirectDraw 画面绘制。这个重要的细节迷惑了许多 DirectX 新手。一般, Windows 给它的子系统如 GDI、MCI 等等发送消息。如果 DirectX 有硬件系统的控制权, 并且共享了它们的独占权, 消息就不被处理。例如, 在全屏幕模式下调用 GDI 图形绘制将被忽略。

但是, 你总可以在子系统下利用软件为你工作, 因为你是一个主管。就像一把的等离子体火焰枪中, 植入了你的 DNA。如果我使用它, 它不发射, 但是如果你拿起它, 它就工作。所以当枪工作时是被使用者独占, 但是枪自身的功能都是始终存在的。不可思议的例子? 哈哈!

因为你在画面进行所有的绘制, 你会设想有办法从 DirectDraw 画面获得 GDI 兼容 DC, 有办法。函数名字是: IDirectDrawSurface4::GetDC(), 如下所示:

```
HRESULT GetDC(HDC FAR *lphdc);
```

你需要做的一切就是调用 DC 的存储器。这不会难倒你。下面是一个例子。

```
LPDIRECTDRAW_SURFACE4 lpdds; // assume this is valid
HDC xdc; // I like calling DirectX DC XDC's
if (FAILED(lpdds->GetDC(*xdc)))
{ /* error */}
// do what you will with the DC...
```

一旦采用了 DirectDraw 兼容 DC, 要像一个正常的 GDI DC 一样, 用后要释放它。函数是 IDirectDrawSurface4::ReleaseDC(), 如下所示:

```
HRESULT ReleaseDC (HDC hDC);
Basically, just sent it the DC you retrieved like this:
if (FAILED(lpdds->Release(xdc)))
{ /* error */}
```

警告



如果画面锁定了, GetDC()就在它上面不起作用了, 因为 GetDC()也锁定画面。另外, 一旦从画面获得 DC, 一定要尽早释放, 因为 GetDC()对画面创建一个内部锁, 你将不能够使用它。基本上, 只有 GDI 或 DirectDraw 中的一个能够在任何时候写画面, 而不是同时。

使用 GDI 的例子, 请参看 DEMO7_17.CPP/EXE。它创建了一个 640×480×256 的全屏幕 DirectDraw 应用, 然后以随机位置, 打印 GDI 文本。打印文字的代码如下所示:

```
int Draw_Text_GDI(char *text, int x, int y,
COLORREF color, LPDIRECTDRAWSURFACE4 lpdds)
{
// this function draws the sent text on the sent surface
// using color index as the color in the palette

HDC xdc; // the working dc

// get the dc from surface
if (FAILED(lpdds->GetDC(*xdc)))
return(0);

// set the colors for the text up
SetTextColor(xdc, color);

// set background mode to transparent so black isn't copied
SetBkMode(xdc, TRANSPARENT);

// draw the text a
TextOut (xdc, x, y, text, strlen(text));

// release the dc
lpdds->ReleaseDC(xdc);

// return success
return(1);
} // end Draw_Text_GDI;GDI;DirectA, combining>
```

技巧



请注意色彩是 COLORREF 格式。这是非常重要的关键点。前面讲过, COLORREF 是 24 位 RGB 结构, 意味着所需色彩总是 24 位 RGB 格式。它的问题是当 DirectX 是在调色模式时, 它就寻找最接近的色彩同所需色彩相匹配, 这一般会导致 GDI 速度变慢, 使得 GDI 打印文字变慢。我强烈要求你对任何要求速度的文字打印, 自己编写文字图形转换器。

函数自己完成所有的 DC 操作, 需要你做的只是调用它。例如, 在组画面用纯绿色在 (100, 100) 处打印 “You da Man!”, 你需要写:

```
Draw_Text_GDI("You da Man!",
100, 100,
RGB(0, 255, 0),
lpddsprimary);
```

在移动之前，我想谈谈何时采用 GDI。通常，GDI 速度较慢，我通常在打印文字、绘制 GUI 材料等等时采用。在发展中一些慢模仿很有用。例如，假设你想写一个实时快速画线算法，叫做 `Draw_Line()`，但是你没有时间来完成，这时可以采用 GDI 的 `Draw_Line()`，这样你至少可以先在屏幕上得到一些东西。等以后你有空了再编写快速的直线绘制算法。

获取 DirectDraw 的真相

正如你所学到的那样，DirectDraw 是一个相当复杂的图像系统。有很多接口，每个接口有大量函数。DirectDraw 的主旨是以一个统一的方式应用硬件。因此，它允许游戏程序员访问 DirectDraw 的不同接口状态和/或能力，使得能够采取合适的动作。例如，你想创建画面，你可能首先想察看 VRAM 有多少可用空间，以便优化创建顺序。你想利用硬件旋转，就应首先知道它是否可用。你想知道的东西可能会很多。因此，在每个主要接口上，都有许多能力测试函数，如 `GetCaps()`、`Get*`()等。让我们来看看最有用的 `GetCaps()` 函数。

DirectDraw 的主对象

DirectDraw 对象自己代表视频卡并描述 HEL 和 HAL。这里有个令人感兴趣的函数叫做 `IDIRECTDRAW4::GetCaps()`：

```
HRESULT GetCaps(
    LPDDCAPS lpDDDriverCaps, // ptr to storage for HAL caps
    LPDDCAPS lpDDHELcaps,   // ptr to storage for HEL caps
```

函数可以重载 HEL 和 HAL。然后，你可以查看有趣的数据返回的 `DDCAPS` 结构。例如，下面是如何获得 HAL 和 HEL：

```
DDCAPS hel_caps, hal_caps;

// initialize the structures
DDRAW_INIT_STRUCT(hel_caps);
DDRAW_INIT_STRUCT(hal_caps);

// make the call
if (FAILED(lpdds->GetDC(&hal_caps, &hel_caps)))
    return(0);
```

此时，你就可以索引 `hel_caps` 或 `hal_caps`，找出有用的东西。`DDCAPS` 看起来形式如下：

```
typedef struct _DDCAPS
```

```

{
DWORD    dwSize;
DWORD    dwCaps;                // driver-specific caps
DWORD    dwCaps2;               // more driver-specific caps
DWORD    dwKeyCaps;             // color key caps
DWORD    dwFXCaps;              // stretching and effects caps
DWORD    dwFXAlphaCaps;         // alpha caps
DWORD    dwPalCaps;             // palette caps
DWORD    dwSVCaps;              // stereo vision caps
DWORD    dwAlphaBlitConstBitDepths; // alpha bit-depth members
DWORD    dwAlphaBlitPixelBitDepths; // .
DWORD    dwAlphaBlitSurfaceBitDepths; // .
DWORD    dwAlphaOverlayConstBitDepths; // .
DWORD    dwAlphaOverlayPixelBitDepths; // .
DWORD    dwAlphaOverlaySurfaceBitDepths; // .
DWORD    dwZBufferBitDepths;    // Z-buffer bit depth
DWORD    dwVidMemTotal;         // total video memory
DWORD    dwVidMemFree;          // total free video memory
DWORD    dwMaxVisibleOverlays;  // maximum visible overlays
DWORD    dwCurrVisibleOverlays; // overlays currently visible
DWORD    dwNumFourCCCodes;      // number of supported FOURCC codes
DWORD    dwAlignBoundarySrc;     // overlay alignment restrictions
DWORD    dwAlignSizeSrc;        // .
DWORD    dwAlignBoundaryDest;    // .
DWORD    dwAlignSizeDest;       // .
DWORD    dwAlignStrideAlign;     // stride alignment
DWORD    dwRops[DD_ROP_SPACE];  // supported raster ops
DWORD    dwReservedCaps;        // reserved
DWORD    dwMinOverlayStretch;    // overlay stretch factors
DWORD    dwMaxOverlayStretch;    // .
DWORD    dwMinLiveVideoStretch;  // obsolete
DWORD    dwMaxLiveVideoStretch;  // .
DWORD    dwMinHwCodecStretch;    // .
DWORD    dwMaxHwCodecStretch;    // .
DWORD    dwReserved1;           // reserved
DWORD    dwReserved2;           // .
DWORD    dwReserved3;           // .
DWORD    dwSVBCaps;             // system-to-video
                                // blit related caps
DWORD    dwSVBCKeyCaps;         // .
DWORD    dwSVBFXCaps;          // .
DWORD    dwSVBRops[DD_ROP_SPACE]; // .
DWORD    dwVSBCaps;            // video-to-system
                                // blit related caps
DWORD    dwVSBCKeyCaps;         // .
DWORD    dwVSBFXCaps;          // .
DWORD    dwVSBRops[DD_ROP_SPACE]; // .
DWORD    dwSSBCaps;            // system-to-system
                                // blit related caps

```

```

DWORD   dwSSBCKeyCaps;           // .
DWORD   dwSSBCFXCaps;           // .
DWORD   dwSSBRops[DD_ROP_SPACE]; //
DWORD   dwMaxVideoPorts;        // maximum number of
                                   // live video ports
DWORD   dwCurrVideoPorts;        // current number of
                                   // live video ports
DWORD   dwSVBCaps2;              // additional
                                   // system-to-video blit caps
DWORD   dwNLVBCaps;             // nonlocal-to-local
                                   // video memory blit caps
DWORD   dwNLVBCaps2;            // .
DWORD   dwNLVBCKeyCaps;         // .
DWORD   dwNLVBFXCaps;          // .
DWORD   dwNLVBRops[DD_ROP_SPACE]; // .
DDSCAPS2 ddsCaps;               // general surface caps
} DDCAPS, FAR* LPDDCAPS;

```

描述每一个字段都需要一本书，所以还是在 SDK 中查找吧。它们大部分都很一目了然，例如 `DDCAPS.dwVidMemFree`，它是我喜欢的成员之一，因为它给出了可以为画面所用的 VRAM 实际的量。

还有一个我喜欢用的函数，即 `GetDisplayMode()`，它给出了在 Windows 下系统的模式。下面是原型：

```
HRESULT GetDisplayMode(LPDDSURFACEDESC2 lpDDsurfaceDesc2);
```

你已经见过 `DDSURFACEDESC2` 结构，应该知道怎么做。

注 意

多数时候，Directx 数据结构可以用来读，也可以写。换句话说，你可以设置数据结构创建对象，但是，当你通过 `GetCaps()` 想了解一个对象时，对象将用它的数据填充同一个数据结构。

在画面上冲浪

很多时候，你不需要找出画面的性能，因为你在创建它时就已经知道了。但是，主画面和后备缓冲画面的性能非常重要，因为它们可以使你洞察每一个硬件的性能。一般画面性能的函数（或者说：“方法”）是：`IDIRECTDRAWSURFACE4::GetCaps()`：

```
HRESULT GetCaps(LPDDSCAPS2 lpDDSCaps);
```

函数返回一个 `DDSCAPS2` 结构，你在以前见过，拿出来利用即可。

下一个与画面相关的函数是 `IDIRECTDRAWSURFACE4::GetSurfaceDesc()`。它返回一

个 DDSURFACEDESC2 结构，它包含画面本身比较重要的信息。下面是原型：

```
HRESULT GetSurfaceDesc(LPDDSURFACEDESC2 lpDDSurfaceDesc);
```

还有一个函数是：DIRECTDRAW_SURFACE4::GetPixelFormat()，我在前面已经谈过，它在结构 DDSURFACEDESC2 中返回像素的格式。

使用调色板

关于使用调色板，要讨论的东西还不多。DirectDraw 仅仅给你一个位编码字，来描述任何给定的调色板的性能。函数是：IDIRECTDRAWPALATTE::GetCaps()，如下所示：

```
HRESULT GetCaps(LPDDWORD lpdwCaps);
```

lpdwCaps 是具有表 7.7 中值的位编码字。

表 7.7 调色板容量可能的标志

值	描 述
DDPCAPS_1BIT	支持 1 位色彩调色板
DDPCAPS_2BIT	支持 2 位色彩调色板
DDPCAPS_4BIT	支持 4 位色彩调色板
DDPCAPS_8BIT	支持 8 位色彩调色板
DDPCAPS_8BITENTRIES	调色板是一个索引调色板
DDPCAPS_ALPHA	支持在每个调色板入口带有一个 alpha 组件
DDPCAPS_ALLOW256	定义了全部 256 种色彩
DDPCAPS_PRIMARYSURFACE	调色板和主画面连接
DDPCAPS_VSYNC	调色板能够随着显示器的刷新同步更新

在窗口模式下应用 DirectDraw

我想讨论的最后一个主题是在窗口模式下应用 DirectDraw。问题是窗口模式，同游戏一样，很难控制色彩深度和分辨率的初始设置。写一个 DirectDraw 全屏幕游戏已经很复杂了，但是采用窗口模式会更加复杂。你必须考虑使用者可能在任何分辨率/或色彩深度的模式下启动游戏。这就意味着你的程序的表现会很糟糕。你可能将游戏设计成只是在 8 位或者 16 位模式下工作，如果玩家使用的是真彩色模式，游戏就根本不能够运行。

虽然让游戏既适用于窗口模式又适用全屏幕模式最好，但是为了减少工作量，我还是常常采用全屏幕模式。但是我可以创建一些窗口应用程序，工作在 8 位色彩深度的 800×600

或者更高的分辨率模式。这样可以方便调试 DirectX 程序或者其他的输出窗口，包括图像输出。下面，我们来看看如何编写 DirectX 窗口程序，如何操作主画面。

要知道的第一件事是窗口 `DirectDraw` 程序采用整个主画面作为整个显示屏幕，而不仅仅是一个窗口，见图 7.38。你不能盲目的向屏幕写操作，否则会破坏其他应用程序窗口。当然，如果你正在写一个屏幕保护程序或者其他的屏幕操纵程序，这正符合你的要求。但是多数时候，你只是想占用自己的用户区窗口。这就意味着你必须找出你的用户区窗口的坐标，并确保只是在它上面操作。

第二个问题是剪切。如果你想使用 `Blit()` 函数，它不管你的用户区，将会图形变换到用户区边缘之外。这就意味着你必须告诉 `DirectDraw`，你在屏幕上有一个窗口，使它进行剪切时，无论你的窗口位置移动或者改变尺寸，都不会出错。

这又产生了另外一个问题，如果窗口被玩家移动或者改变尺寸怎么办？当然，你可以强制窗口的大小，但是移动绝对需要。另外，为什么要有窗口应用程序？处理这些问题，必须跟踪 WM_SIZE 和 WM_MOVE 消息。

下一个问题是 8 位调色板。如果视频是 8 位，你想改变调色板就会有问题。你必须调整 Windows 的调色板管理器，否则，其他的应用程序会变得很难看。你可以改变调色板，但是如果留下大约 20 个调色板入口（Windows 和系统色彩），使调色板管理器能够让其他应用程序正常工作，会是一个更好的主意。



图 7.38 在窗口模式下, 整个桌面映射到 DirectDraw 主画面

最后，最明显的问题是如何在不同的模式下，进行图形变换和像素操作。因此，如果你希望完美一些的话，就一定要写代码处理所有的色彩深度。

所以，我将从认识窗口模式下的细节开始。你已经知道如何做这点。不同的是你没有

设过视频模式或者创建备用后备缓冲。在窗口模式下，不允许你进行页交换。你必须图形变换到双缓冲或者自己完成它，但是你不能建立复杂的画面链然后调用 Flip()。它不会起作用。事实上这也没有问题，你可以在缓冲中创建一个同用户区一样大小的另一个画面，在它的上面绘制，然后把它图形变换到主画面上窗口中的用户区。通过这样，你可以避免屏幕闪烁。

让我给你演示创建 DirectDraw 窗口应用程序的过程，首先，进行 DirectDraw 的初始化。

```
LPDIRECTDRAW lpdd_temp    NULL; // used to get IDirectDraw4 interface

// first create base IDirectDraw interface
if (FAILED(DirectDrawCreate(NULL, &lpdd_temp, NULL)))
    return(0);

// now query for IDirectDraw4
if (FAILED(lpdd_temp->QueryInterface(IID_IDirectDraw4,
                                     (LPVOID *)&lpdd)))
    return(0);

// set cooperation to full screen
if (FAILED(lpdd->SetCooperativeLevel(main_window_handle, DDSCL_NORMAL)))
    return(0);

// clear ddsd and set size
DDRAW_INIT_STRUCT(dds);

// enable valid fields
dds.dwFlags = DDSCL_CAPS;

// request primary surface
dds.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// create the primary surface
if (FAILED(lpdd->CreateSurface(&dds, &lpddsprimary, NULL)))
    return(0);
```

下面最主要的是合作等级的设置。注意，它是 DDSCL_NORMAL，而不是通常在全屏模式下的 (DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX | DDSCL_EXCLUSIVE | DDSCL_ALLOWREBOOT)。另外，你在创建窗口应用程序时，使用 WS_OVERLAPPED 或者 WS_OVERLAPPEDWINDOW，而不是 WS_POPUP。WS_POPUP 创建的窗口没有任何标题控制等等。WS_OVERLAPPED 创建的窗口有标题，但是不能改变尺寸，WS_OVERLAPPEDWINDOW 格式创建带有全部功能的窗口。但是，在多数时候，我采用 WS_OVERLAPPED，因为我不想处理窗口尺寸的改变，用哪一种取决于你。

要更深入地了解这点，可参考演示程序 DEMO7_18.CPP/EXE。它创建了一个 DirectX

的应用程序，具有一个带主画面的 400×400 窗口。

在窗口中绘制像素

现在，让我们回到利用窗口的用户区。记住两件事情：主画面是整个屏幕，你不知道色彩深度。首先看第一个问题——找出窗口的用户区。

因为玩家可以将窗口移动到屏幕的任何地方，如果想使用绝对坐标的话，用户区的坐标总是变化的，你必须找出窗口的用户区左上角在屏幕上的坐标，然后利用它作为原点绘制像素。你需要的函数是（我在以前谈到过）`GetWindowRect()`：

```
BOOL GetWindowRect(HWND hwnd,    // handle of window
LPRECT, lpRect); // address of structure
// for window coordinates
```

警告



`GetWindowRect()` 实际上重载整个窗口的坐标，包括边缘和控制。我将教给你如何找出用户区的坐标，但是记住上面那点……

当你发送应用程序窗口的窗口句柄时，函数在 `lpRect` 中返回你的窗口的用户区坐标。所以，所有你需要做的就是调用这个函数，重载你窗口左上角在屏幕上的坐标。当然，窗口总是在移动，坐标在变化，所以，每一帧或者收到一个 `WM_MOVE` 消息时你都需要调用函数 `GetWindowRect()`。我喜欢每帧调用一次，因为我讨厌处理 Windows 消息。

现在既然知道了你的窗口的用户区屏幕坐标，准备进行像素操作吧。但是，等等，像素格式是什么？

很高兴你能问这个问题，因为我知道你一定知道答案。你需要首先调用 `GetPixelFormat()` 函数，决定色彩深度等等。基于此，调用不同的像素绘制函数。所以，在你程序的一个地方，也许是在设置 `DirectDraw` 之后在 `GameInit()` 函数中，应该从主画面调用 `GetPixelFormat()`，像下面这样。

```
int pixel_format = 0; // global to hold the bpp
DDPIXELFORMAT ddpixelfomat; // hold the pixel format

// clean out the structure and set it up
DDRAW_INIT_STRUCT(ddpixelfomat);

// get the pixel format
lpddsprimary->GetPixelFormat(&ddpixelfomat);

// set global pixel format
pixel_format = ddpixelfomat.dwRGBBitCount;
```

然后，一旦你知道了像素的格式，就可以利用逻辑坐标、函数指针或虚函数设置像素绘制函数的正确色彩深度。为了保持简单，你可以仅仅利用一些坐标逻辑，在绘制之前检测全局变量 `pixel_format`。下面是在窗口的用户区的随机位置绘制随机像素的代码：

```
DDSURFACEDESC2 ddsd;    // directdraw surface description
RECT client;             // used to hold client rectangle

// get the window's client rectangle in screen coordinates
GetWindowRect(main_window_handle, &client);

// initialize structure
DDRAW_INIT_STRUCT(dds);

// lock the primary surface
lpddsprimary->Lock(NULL,&dds,
                  DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT,NULL);

// get video pointer to primary surface
// cast to UCHAR * since we don't know what we are
// dealing with yet and it's like bytes :)
UCHAR *primary_buffer = (UCHAR *)dds.lpSurface;

// what is the color depth?
if (pixel_format == 32)
{
    // draw 10 random pixels in 32 bit mode
    for (int index=0; index<10; index++)
    {
        int x=rand()%(client.right - client.left) + client.left;
        int y=rand()%(client.bottom - client.top) + client.top;
        DWORD color = _RGB32BIT(0,rand()%256, rand()%256, rand()%256);
        *((DWORD *) (primary_buffer + x*4 + y*dds.lPitch)) = color;
    } // end for index
} // end if 24 bit

else
if (pixel_format == 24)
{
    // draw 10 random pixels in 24 bit mode (very rare???)
    for (int index=0; index<10; index++)
    {
        int x=rand()%(client.right - client.left) + client.left;
        int y=rand()%(client.bottom - client.top) + client.top;
        ((primary_buffer + x*3 + y*dds.lPitch))[0] = rand()%256;
        ((primary_buffer + x*3 + y*dds.lPitch))[1] = rand()%256;
        ((primary_buffer + x*3 + y*dds.lPitch))[2] = rand()%256;
    } // end for index
} // end if 24 bit
```

```

else
if (pixel_format == 16)
{
// draw 10 random pixels in 16 bit mode
for (int index=0; index<10; index++)
{
int x=rand()%(client.right - client.left) + client.left;
int y=rand()%(client.bottom - client.top) + client.top;
USHORT color = _RGB16BIT565(rand()%256, rand()%256, rand()%256);
*((USHORT *) (primary_buffer + x*2 + y*ddsd.lPitch)) = color;
} // end for index
} // end if 16 bit
else
{ // assume 8 bits per pixel
// draw 10 random pixels in 8 bit mode
for (int index=0; index<10; index++)
{
int x=rand()%(client.right - client.left) + client.left;
int y=rand()%(client.bottom - client.top) + client.top;
UCHAR color = rand()%256;
primary_buffer[x + y*ddsd.lPitch] = color;
} // end for index
} // end else

// unlock primary buffer
lpddspriamry->Unlock(NULL);

```

技巧

当然，这段代码尚未优化，我极不情愿给你如此之慢而且原始的代码，但是它好理解。在实际应用时，你应该使用函数指针、虚函数，完全去掉重复部分和多余部分，使用递增寻址。这也算是安慰一下我。

演示程序 DEMO7_19.CPP1EXE 绘制了任意色彩深度的像素。它创建了一个 400×400 的窗口，然后在用户区绘制像素。试着在不同色彩模式下运行程序，注意它仍然工作。做完后，回来，我将谈谈在实际的内部用户区精确着色问题。

查找真实的用户区

采用窗口的问题是：当你采用 CreateWindow() 或者 CreateWindowEx() 函数创建窗口时，你要指定窗口尺寸的总体宽度，包括各种控件。因此，如果创建的窗口是一个没有任何控件的空白 WS_POPUP 窗口，它的整个尺寸就都是用户区尺寸。另外，如果采用 CreateWindowEx() 创建窗口，窗口尺寸就会缩小，以放置控件、菜单、标题等等。结果是工作区比你要求的要小。图 7.39 显示了这种情况。解决办法是重新考虑你的窗口尺寸，包括边缘、控件等等。

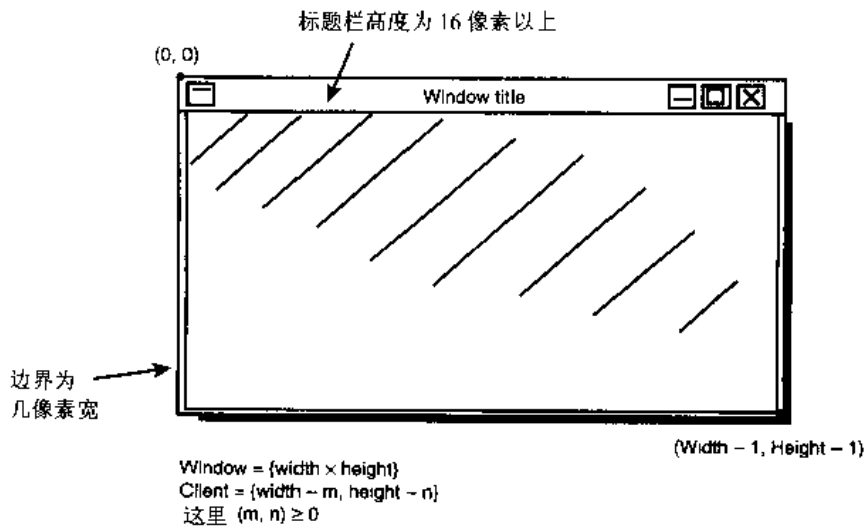


图 7.39 窗口用户区小于包围它的窗口

例如，假设你想有一个 640×480 的工作区，但是又想有控件、边界、菜单等等，这时需要你重新计算额外的窗口附件在 X、Y 方向上分别需要多少像素，增加你的窗口的尺寸，直到用户区的大小正符合你的要求为止。计算不同窗口样式下尺寸的魔术般的函数是 `AdjustWindowRectEx()`，如下所示。

```

BOOL AdjustWindowRectEx(
    LPRECT lpRect,      // pointer to client-rectangle structure
    DWORD dwStyle,      // window styles
    BOOL bMenu,         // menu-present flag
    DWORD dwExStyle);   // extended style

```

填充各个参数，函数会得出整个窗口标志和样式，调整发过来的 `lpRect` 结构数据。使用该函数，首先设置矩形为你想要的用户区的大小，即 640×480 。然后，用所创建的原始窗口的所有合适参数调用函数。但是，我从来没能记住设置的窗口是什么样，或者标志的正确名字，所以你可以让 `windows` 告诉你已经做了什么，保存了什么。下面是调用情况，同时有窗口的帮助函数来查询你发送的基于 `HWND` 的样式：

```

// the client size we desire
RECT window_rect = {0, 0, 640, 480};
// make the call to adjust window_rect
AdjustWindowRectEx(&window_rect,
    GetWindowStyle(main_window_handle),
    GetMenu(main_window_handle) != NULL,
    GetWindowExStyle(main_window_handle));

// now resize the window with a call to MoveWindow()
MoveWindow(main_window_handle,

```

```

CW_USEDEFAULT, // x position
CW_USEDEFAULT, // y position
Window_rect.right - window_rect.left, // width
Window_rect.bottom - window_rect.top, // height

```

就是这样！

剪切 DirectX 窗口

现在已经有些眉目了 IDirectDraw 的窗口模式的下一个难题是使用剪切。你一定要理解，剪切只是和图形变换器有关，它对你直接采用的主画面不起作用。

你在以前看到了 IDIRECTDRAWCLIPPER 接口，所以现在我就不多讲它了（我仍然为在 RECT 中的正确坐标而激动）。需要你做的第一件事是创建一个 DirectDraw 剪切板。就像下面这样：

```

LPDIRECTDRAWCLIPPER lpddclipper = NULL; // hold the clipper
if (FAILED(lpdd->CreateClipper(0, &lpddclipper, NULL)))
return(0);

```

接着，你必须将剪切板用函数 IDIRECTDRAWCLIPPER::SetHWND() 同窗口连接。这使得剪切板同你的窗口联系在一起，并为你处理所有的细节，如尺寸移动等。甚至你都不用发送剪切序列，它是自动的。函数十分简单，原型如下：

```

HRESULT SetHWND(DWORD dwFlags, // unused, set to 0
HWND hWnd); // app window handle

```

下面是调用连接你的剪切板给你的主窗口。

```

if (FAILED(lpddclipper->SetHWND(0, main_window_handle)))
return(0);

```

接下来，你需要将剪切板同想要剪切的画面相连接——这里指主画面。为实现这点可使用 SetClipper()，你也见过。

```

if (FAILED(lpddsprimary->SetClipper(lpddclipper)))
return(0);

```

警告



现在就大功告成了，但是，还有一个 DirectX 的小问题。创建剪切板参考计数器为 2，1 是创建，2 是调用 SetClipper()。这很好，但是，销毁表面时，并不销毁剪切板，也就是说，你必须在调用 lpddsprimary->Release() 销毁表面之后，调用 lpddclipper->Release 销毁剪切板。问题是你可能认为通过 lpddclipper->Release() 就消除了剪切板，但是，它只是释放了第一步。微软建议你在前面的代码之后，马上调用 lpddclipper->Release()，使得参考计数器 lpddclipper 为 1，它应该是这样。

记住，连接到窗口的剪切板只是对主画面图形变换，也就是说窗口的内容。但是，许多时候，会创建一个备用画面模拟双缓冲，图形变换它，然后采用图形变换器——一个原始的页交换方法——将它拷贝到主画面。如果你图形变换主画面超出了范围，将图形变换器同主画面连接才有帮助。这只有在使用者改变尺寸的时候才会发生。

以 8 位窗口模式工作

我想介绍的最后一个问题是 8 位窗口模式和调色板。简而言之，你只是可以创建一个调色板，在它上面完成想做的事，同主画面相连。你一定用过一点 Windows 调色板管理器。在 GDI 下的 Windows 调色板管理器超出了本书的范围（我总想这么说），所以我不想用它的繁琐的细节搞糊涂你。问题是，如果你想在窗口模式，256 色下运行你的游戏，受你支配的色彩实际上不到 256 色。

每个在桌面上运行的应用程序都有一个逻辑调色板，存放着要用的色彩。但是，物理调色板实际上只有一个。物理调色板反映实际的硬件调色板，这是 Windows 工作时的折衷。当你的应用程序想聚焦时，你的逻辑调色板被激活，Windows 调色板管理器将你的色彩以最大程度映射到物理调色板。有时工作得很好，但有时就不。

另外，你设置的逻辑调色板的标志决定 Windows 工作的宽松程度。至少 Windows 需要在调色板之外有 20 色：前 10 个和后 10 个。这些是 Windows 保留的最少色彩，负责使 Windows 应用程序看起来正常。创建 256 色模式下的 Windows 应用程序的技巧是将你的艺术品限制在 236 色之内——给 Windows 留下色彩空间——然后再设置你的逻辑调色板的标志。当你的调色板被激活后，Windows 就不再管它们。下面是创建一个一般调色板的代码。你可以将代码修改为你的 RGB 色彩入口为 10~245。代码只是使它们变暗：

```
LPDIRECTDRAW4 lpdd; // this is already setup
PALETTEENTRY palette[256]; // holds palette data
LPDIRECTDRAWPALETTE lpddpal=NULL; // palette interface

// first set up the windows static entries
// note it's irrelevant what we make them
for (int index = 0; index<10; index++)
{
    // the first 10 static entries
    palette[index].peFlags = PC_EXPLICIT;
    palette[index].peRed = index;
    palette[index].peGreen = 0;
    palette[index].peBlue = 0;

    // the last 10 static entries
    palette[index+246].peFlags = PC_EXPLICIT;
    palette[index+246].peRed = index+246;
    palette[index+246].peGreen = 0;
    palette[index+246].peBlue = 0;
}
```

```

    } // end for index

    // now set up our entries. You would load these from
    // a file etc., but for now we'll make them grey
    for (int index = 10; index < 246; index++)
    {
        palette[index].peFlags = PC_NOCOLLAPSE;
        palette[index].peRed = 64;
        palette[index].peGreen = 64;
        palette[index].peBlue = 64;

        // Create the palette
        if (FAILED(lpdd->CreatePalette(DDPACAPS_8BIT, palette,
        & lpddpal, NULL)))
        { /* error */}

        // attach the palette to the primary surface...

```

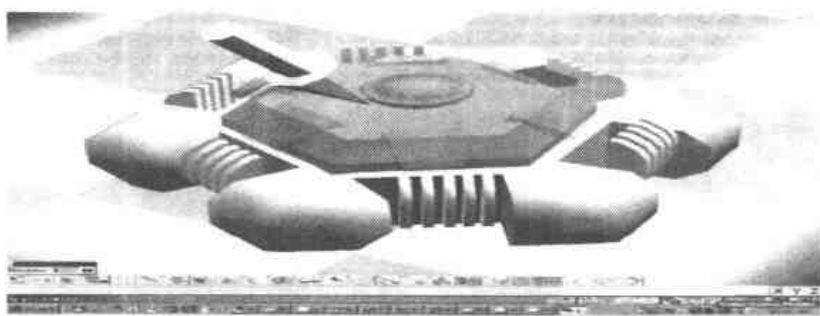
注意 PC_EXPLICIT 和 PC_NOCOLLAPSE 的用法。PC_EXPLICIT 意味着“这些色彩映像到硬件”，PC_NOCOLLAPSE 意味着“不要将色彩映像到任何入口，就让它们那样”。如果想要使色彩寄存器运动，你需要逻辑“或”标志 PC_RESERVED。这就通知调色板管理器，不要让其他 Windows 应用程序色彩映像到入口，因为在任何时候它都可能改变。

总 结

无疑，这是本书中最长的一章，嘿！原谅我。要讨论的内容太多了，我甚至可以继续，但是，我还想在雨林中留下一些树。

我覆盖了所有内容，包括真彩色模式、图形变换、剪切、色彩关键字、采样理论、窗口 DirectDraw、GDI 及从 DirectDraw 中获取信息，我还用了 T3DLIB1.CPPH 库函数。如果你愿意，你可以现在就看看它，但是我将在下一章再给你讲解每一个函数。

下一章，我们将离开 DirectDraw 休息一会（虽然在该章的末尾又出现了），开始探讨一些二维几何的矢量技术、移动技术和光栅化理论。



8

矢量光栅化及 2D 变换

本章将讨论矢量的问题，如：如何画直线和多边形，并演示如何建立第一个 T3DLIB 游戏库。本章是数学讨论的第一章，不过，没有关系，只要花费一点点时间加上少许努力，你就会发现没有掌握不了的。在我所讲的内容中最难的部分是矩阵（仅仅是为了使您在 3D 中不至于对矩阵陌生而做的简介）。当然，根据多数人的需要，本章将就滚动、等角 3D 引擎给出一些有用的建议。记住，这是一本关于 3D 的书，所以本书节省了其他方面的篇幅留给了 3D。但是，文中没有介绍的部分您会在 CD 中学到。这里列出本章的主要内容。

- ✦ 画线
- ✦ 剪切
- ✦ 矩阵
- ✦ 2D 变换
- ✦ 画多边形
- ✦ 滚动和等角 3D 引擎
- ✦ 计时器
- ✦ T3DLIB 1.0

绘制线条

到目前为止，你所画的对象是由点或者由位图构成。这两种实体都不是矢量实体（矢量实体如图 8.1 中的直线或者多边形）。所以，你的第一个难题是如何进行直线的绘制。

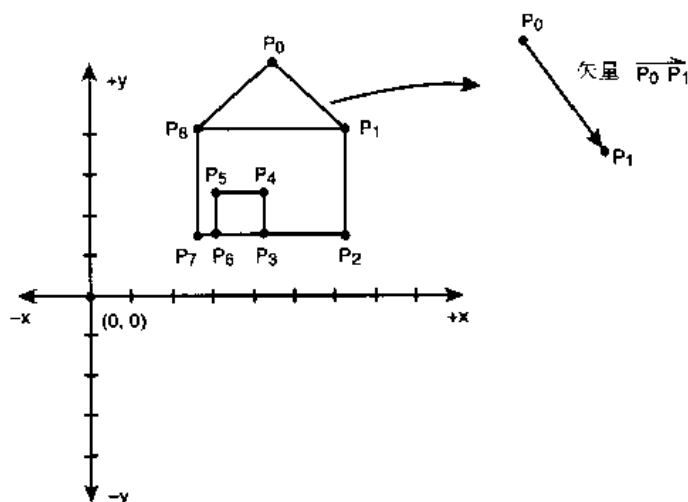


图 8.1 基于矢量直线的对象

你可能会认为画直线微不足道，但我向您保证，事实并非如此。在计算机的屏幕上画直线有一系列问题，如有限的分辨率、实数向整数网格的映射、速度等等。许多时候，2D 或 3D 图形是由一些构成对象或场景的面或多边形组成，而这些面或多边形又由许许多多的点组成。这些点通常表现为实数形式，如点 (10.5, 120.3) 等。

第一个问题是：由于计算机屏幕上的点是由一个整数组成的网格所控制，因此要在屏幕上显示点 (10.5, 120.3) 根本不可能。你只有采用近似值。你可能把该点显示为 (10, 120)，或者显示为 (11, 120)。最后，你还可能采用高技术，通过像素画点函数画出像素中心在 (10.5, 120.3) 的、具有不同像素浓度的一系列点来代替这一点（见图 8.2）。

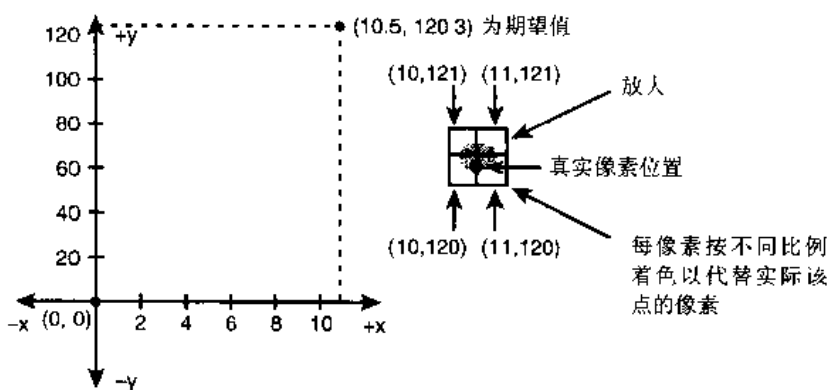


图 8.2 过滤单像素的区域

基本上，一个区域过滤器来计算需要多少原始像素迭加成一个另外位置的像素位置，然后使这些像素浓度低于原始浓度，而像素颜色采用原始颜色，画出这些像素点。这样可以画出一个看起来没有走样的圆点，但同时你的最大分辨率也降低了。

现在，不再在直线画图的运算、过滤上喋喋不休，让我们来学习两种真正很好的、实用的画直线的运算方法，用这两种算法，你可以很好地绘制出一个坐标从 (x_0, y_0) 到 (x_1, y_1) 的直线。

Bresenham 算法

下面要介绍的第一个算法被称为 Bresenham 算法，它是由 Bresenham 在 1965 年发明的。起先算法的发明是为了在绘图仪上画线，后来才被用作计算机绘图。首先让我们浏览一下该算法的工作过程，然后再看一些代码。图 8.3 给出了通常需要解决的问题，你需要把像素以最接近实际直线的位置从点 p_1 到 p_2 进行填充。这个过程被称为光栅化。

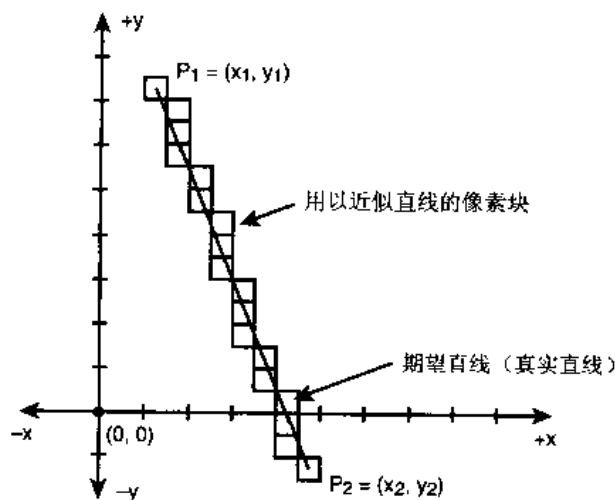


图 8.3 对一条直线进行光栅化

如果你对直线的概念忘却了，下面我来提醒你。直线的斜率同直线与 x 轴的夹角有关，直线的斜率为 0，则该直线为水平直线；直线的斜率为无穷大，则直线为垂直直线；直线的斜率为 1；则为 45 度的对角斜线。斜率的定义为单位前进距离的上升量，数学表达式为：

$$\text{slope} = \frac{\text{Rise}}{\text{Run}} = \frac{\text{Change in } y}{\text{Change in } x} = \frac{dy}{dx} = m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

比如：若直线过点 $p_0(1, 2)$ 和点 $p_1(5, 22)$ ，则直线的斜率 m 为：

$$(y_2 - y_1) / (x_2 - x_1) = (22 - 2) / (5 - 1) = 20/4 = 5$$

那么，斜率的物理意义意味着什么呢？它意味着 x 坐标增加 1 个单位， y 坐标就增加 5 个单位。OK，这是网格算法的一个开始，下面，给出画线思想一个大概轮廓。

1. 计算斜率 m 。
2. 画点 (x_0, y_0) 。
3. 在 x 方向每前进 1 个单位，在 y 方向就增加 5 个单位。
4. 重复 2~4 步，直到结束。

图 8.4 给出了点 p1 到点 p2 绘制的示意图。

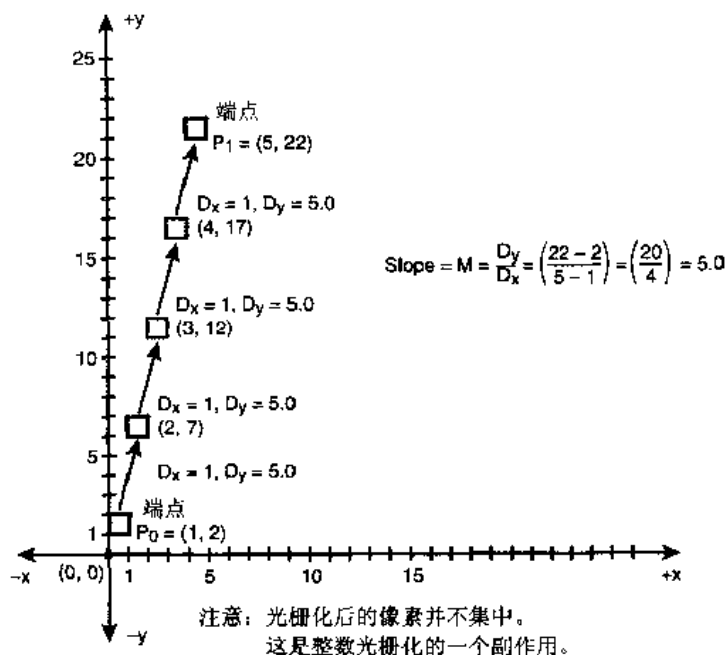


图 8.4 光栅化的第一步

现在你明白了问题所在吗？在 x 方向每前进 1 个单位，在 y 方向就要前进 5 个单位，在你所画的线上留下许多洞，这使得你所画的线不是连续的线，而是有间隔的一些像素点。也就是说，无论何时， x 总是一个整数。基本上，你所填充的是在直线上的一些整数点，该直线可表达为：

$$(y-y_0) = m (x-x_0)$$

这里， (x, y) 为当前点，或者称为像素坐标， (x_0, y_0) 是起始点， m 是斜率。整理公式可得：

$$y = m (x - x_0) + y_0$$

因此，如果在本式中 (x_0, y_0) 为 $(1, 2)$ ；斜率 $m=5$ ，则你可以得到以下结果

x	$y=5(x-1)+2$
1	2 (起点)
2	7
3	12
4	17
5	22 (终点)

现在，你可能会问，下面的直线的斜率-截距表达式是否有用

$$y=mx+b$$

这里， b 是直线在 y 轴上的截距。可以，但是，这对你没有什么好处。根本的问题在于你每步在 x 轴方向移动 1 个单位。你移动的量必须非常小，为了使它能够捕获每一个像素，有时步长只能是 0.01，否则你就不得不尝试其他方法。聪明的读者会意识到，无论你采取多小的 x 步长，你都有可能找到一个有跳过的点的斜线，因此不得不采用其他方法，这就是 Bresenham 算法的基础。

在计算机解释命令中，Bresenham 算法开始于 (x_0, y_0) 点，但是并不是沿着斜率进行画线，它是先在 x 方向上移动一个像素，然后再决定 y 方向像素的移动，移动 y 像素的原则是使得线的轨迹尽量接近于实际的线。在使所画的线的光栅位置尽量接近于实际线的过程中伴随着误差项。算法对误差项不断调整，以使得数字化的光栅线能够尽量接近实际。

算法一般在笛卡尔坐标系中的第一象限，其他象限通过映像来获得。另外，算法考虑了两种斜率的线，一种是斜率小于 45 度的线，即 $m < 1$ 的线和斜率大于 45 度，即 $m > 1$ 的线。我更喜欢分别将它们称为 x 主线或 y 主线。下面是从 $p_0(x_0, y_0)$ 到 $p_1(x_1, y_1)$ x 主线的伪代码。

```
// initialize starting point
x=x0;
y=y0;

// compute deltas
dx=x1-x0;
dy=y1-y0;

// initialize error term
error = 0;

// draw line
for (int index = 0; index < dx; index++)
{
    // plot the pixel
    Plot_Pixel(x, y, color);

    // adjust the error
    error+=dy;

    // test the error
    if (error > dx)
    {
        // adjust error
        error-=dx;

        // move up to next line
        y++;
    } // end if
} // end for index
```

当然，上面只是整个坐标系中 45 度角以内的线的算法，但是简单地改变一下符号和变

换数值，所有其他线都可以画出，算法是一致的。

在给出你代码之前，我想给你指出一点，就是精度问题。算法不停地减小光栅线和实际线之间的误差，但是初始点还可以选取得更好一些。你看到，开始的误差值为 0.0。这实际上是不正确的，如果在开始时采用一个位于最小误差和最大误差之间的一个误差值，情况会更好一些。这可以通过将初始误差设为 0.5 来做到，但是由于你现在使用的是整数量，所以你必须将它乘以 2，然后再将之加到 dx 或 dy 分量上。最终结果是你要改写算法，其误差项如下所示：

```
//x-dominate
error=2*dy-dx

//y-dominate
error=2*dx-dy
```

这样一来，你便将误差相应增大了两倍。这里给出最终的算法，它包含在你的函数库里，名字为 Draw_Line()。注意该算法采用了两个端点、颜色、视频缓冲、视点间距，然后再画线。

```
int Draw_Line(int x0, int y0, // starting position
              int x1, int y1, // ending position
              UCHAR color,    // color index
              UCHAR *vb_start,
              int lpitch)     // video buffer and memory pitch
{
    // this function draws a line from x0,y0 to x1,y1
    // using differential error
    // terms (based on Bresenham's work)

    int dx,          // difference in x's
        dy,          // difference in y's
        dx2,         // dx,dy * 2
        dy2,
        x_inc,       // amount in pixel space to move during drawing
        y_inc,       // amount in pixel space to move during drawing
        error,       // the discriminant i.e. error i.e. decision variable
        index;       // used for looping

    // precompute first pixel address in video buffer
    vb_start = vb_start + x0 + y0*lpitch;

    // compute horizontal and vertical deltas
    dx = x1-x0;
    dy = y1-y0;

    // test which direction the line is going in i.e. slope angle
    if (dx>=0)
```



```

    {
        x_inc = 1;

    } // end if line is moving right
else
    {
        x_inc = -1;
        dx    = -dx; // need absolute value

    } // end else moving left

// test y component of slope

if (dy>=0)
    {
        y_inc = lpitch;
    } // end if line is moving down
else
    {
        y_inc = -lpitch;
        dy    = -dy; // need absolute value

    } // end else moving up

// compute (dx,dy) * 2
dx2 = dx << 1;
dy2 = dy << 1;

// now based on which delta is greater we can draw the line
if (dx > dy)
    {
        // initialize error term
        error = dy2 - dx;

        // draw the line
        for (index=0; index <= dx; index++)
            {
                // set the pixel
                *vb_start = color;

                // test if error has overflowed
                if (error >= 0)
                    {
                        error-=dx2;

                        // move to next line
                        vb_start+=y_inc;

                    } // end if error overflowed

                // adjust the error term

```

```

        error+=dy2;

        // move to the next pixel
        vb_start+=x_inc;

    } // end for

} // end if !slope| <= 1
else
{
    // initialize error term
    error = dx2 - dy;

    // draw the line
    for (index=0; index <= dy; index++)
    {
        // set the pixel
        *vb_start = color;

        // test if error overflowed
        if (error >= 0)
        {
            error-=dy2;

            // move to next line
            vb_start+=x_inc;

        } // end if error overflowed

        // adjust the error term
        error+=dx2;

        // move to the next pixel
        vb_start+=y_inc;

    } // end for

} // end else !slope| > 1

// return success
return(1);

} // end Draw_Line

```

注 意

此函数工作在 8 位模式下，但在函数库中有 16 的位版本，名字为 Draw_line16()。

函数基本可分为3部分。第一部分对符号和端点交换，并计算 x 、 y 轴增量。然后根据线的主要走向，即根据 $dx > dy$ ，或 $dx \leq dy$ ，分别进行两个循环的运算画线。

提高算法的速度

看到代码，你可能会认为太严密，没有进行优化的空间。但是有两种方法可以优化算法。第一种方法是考虑到所有的直线都关于它们的中点对称，如图 8.5 所示，所以没有必要对整条直线进行运算。需要你做的是将整条直线一分为二，并复制另一半直线。理论上这很简单，但实际上是当直线的点数为奇数时，你将不得不考虑将这额外的一点画在哪一边。与其说困难不如说可恶。

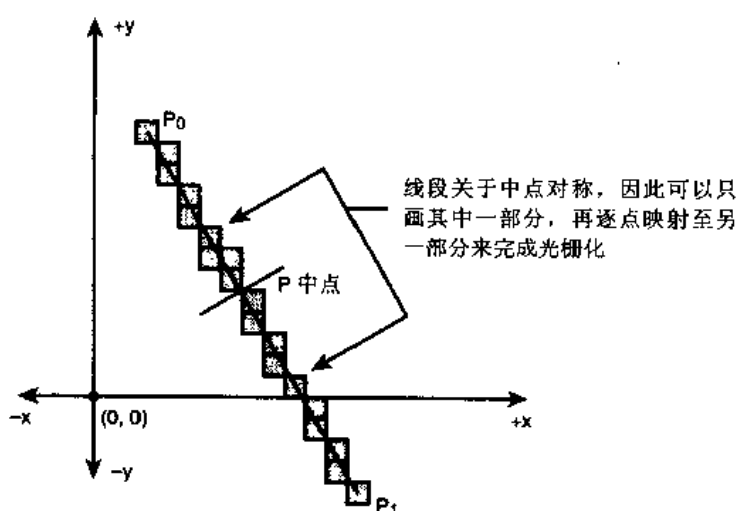


图 8.5 直线关于其中点对称

另外一种优化方法被许多人发现（包括本人），如 Machael Abrash 的 Run-Slicing 算法、Xialon Wu 的 Symmetric Double Step 算法和 Rokne 的 Quadruple Step 算法。所有这些算法的基本点都是利用像素的连续性画线。Run-Slicing 算法是建立在存在线中的连续的多像素基础之上的，如线 $(0, 0) \sim (100, 1)$ 。此线包含两个连续多像素段，一个是 $(0 \sim 49, 0)$ ，另一个是 $(50 \sim 100, 1)$ ，如图 8.6 所示。

那么，在算法中，计算每个像素位置的根本点是什么呢？问题是虽然算法的内部结构非常复杂，但是它适用于计算出具有很大大或很小斜率的斜线。Xialon Wu 的 Symmetric Double Step 算法也采用同样的原理，但是它考虑的不是线的连续性，而是考虑到每两点之间，只有四种联系模式，如图 8.7 所示。采用这种方法根据误差项很容易计算出下一种联系模式，以及整个的画线模式。基本上是以正常速度的两倍进行。因此，再考虑到对称，此方法将基本算法的速度提高了四倍。

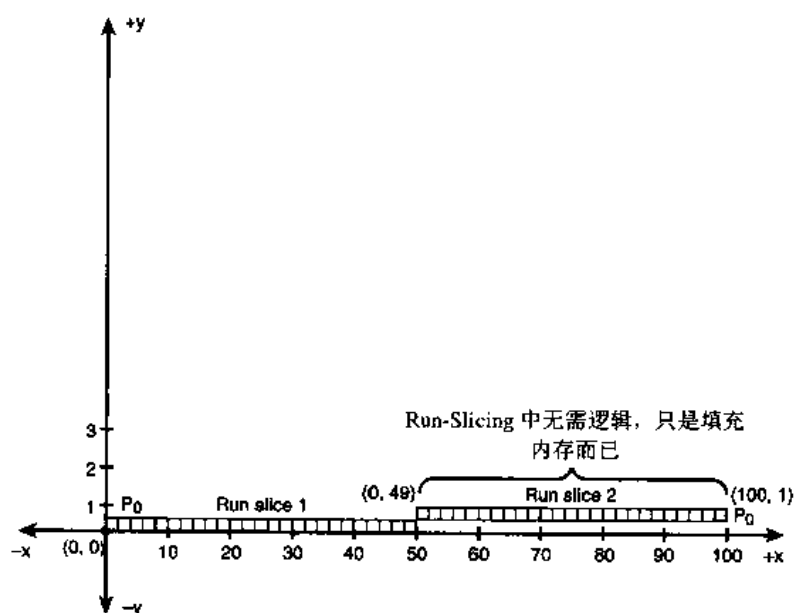


图 8.6 Run-Slicing 直线绘制中的优化

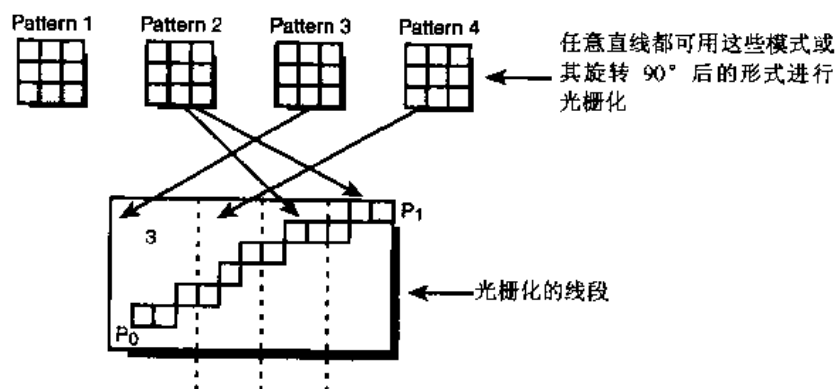


图 8.7 使用光栅的方式绘制直线

注意

如果我认为你需要一种快的画线方式的话，我会把它写出来。但是现在我更喜欢简洁，所以让我们来看剪切吧。

DEMO8_1.CPP1EXE 就是以这种画线方式演示的画线程序，画的是 640×480 模式下的随机线。

基本 2D 图形的剪切

有两种计算机图形剪切方式：图像空间方式和对象空间方式。图像空间方式实际上是像素点模式，图像被光栅化之后，由一个过滤器来决定某一个像素点是否在视区之内。这适合于画单个像素，但不适合于画大的对象，如位图、直线和多边形。对对象来说它们都有一定的集合形状，你可以利用额外的信息。

例如，你想写一个位图剪切程序，你所作的是由一个方框剪切另外一个方框。也就是说，将位图剪切成四边形边界放到你的视野中。两个方框的相交部分就是你所需要的部分。

你可能不信，直线更难剪切。看图 8.8 你就会发现将直线剪切到四边形视区中存在的问题。

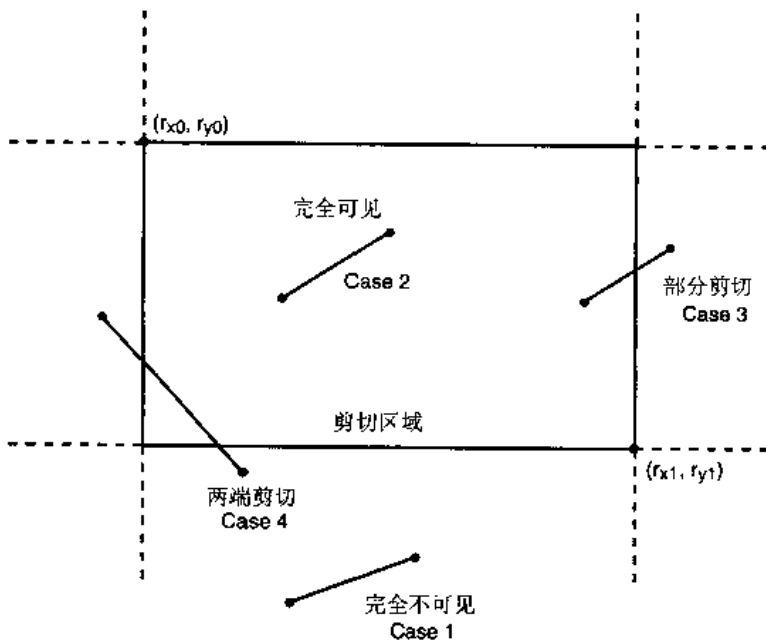


图 8.8 一般直线剪切中的问题

就像你所看到的那样，剪切中存在 4 种情况。

1. 直线完全在剪切区之外。这种情况正符合我们的要求，非常好。
2. 直线完全在剪切区之内。这种情况下，无需对直线进行任何改动，直线可保持原来的光栅化状态。
3. 直线的一部分在剪切区之外，剪切区部分必须进行剪切。
4. 直线的两个端点都在剪切区之外，必须进行剪切。

有许多算法进行直线的剪切，如 Cohen-Sutherland、Cyrus-Beck 等。但是，在你参考别人的算法之前，看你是否自己能够设计出来！

假设你在 2D 空间有一条从点 (x_0, y_0) 到 (x_1, y_1) 的直线，同时又有一个矩形剪切

区由点 (rx_0, ry_0) 、 (rx_1, ry_1) 来限定。你想剪切所有的直线到该剪切区。这时你需要一个预处理器——如果愿意，也可以称为剪切过滤器——处理输入的 (x_0, y_0) 、 (x_1, y_1) 、 (rx_0, ry_0) 、 (rx_1, ry_1) 值。并且输出一个新的线段，从 (x'_0, y'_0) 到 (x'_1, y'_1) ，这代表所产生的新线。本过程如图 8.9 所示。看到这里，你应该首先注意的事情是在有些区域，你不得不计算两条线的相交部分。这是一个你必须解决的基本问题，所以你可以从这里出发。

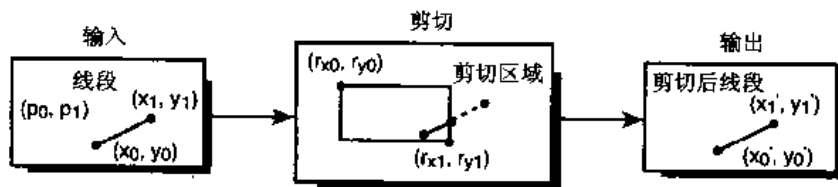


图 8.9 剪切过程图解

通常，线段 (x_0, y_0) (x_1, y_1) 将同剪切区矩形的上、下、左、右边中的一个相交。这意味着你不必找出两条任意方向直线的相交部分，因为光栅化后的直线相交部分只有水平或垂直两种情况。知道这点可能有用也可能无用，因为它并不重要。计算两条直线的交点，有很多种方法，但都是基于以下的直线数学表达式。

一般情况下，直线采用如下表达式。

截距式： $y=mx+b$

点斜式： $(y-y_0)=m(x-x_0)$

两点式： $(y-y_0)=(x-x_0)(y_1-y_0)/(x_1-x_0)$

普通式： $ax+by=c$

参数式： $p=p_0+Vt$

提示

如果你对这些公式不熟悉，不用担心，我简单解释一下带参等式，注意一下会发现，点斜式和两点式其实是一种，因为在两个公式中，都有 $m=(y_1-y_0)/(x_1-x_0)$

采用点斜式计算两条直线的交点

我从心眼里喜欢点斜式和普通式。作为代数的一个很好的例子，让我们算完两种直线的交点，分别由 p_0 、 p_1 来表示。这只是对你进行一下热身，因为我们将要在后面的章节中遇到真正困难的数学。先让我们来做一些普通的点斜式（如图 8.10 所示）。

假设第一条线段为 p_0 为： (x_0, y_0) 到 (x_1, y_1)

假设第二条线段 p_1 为： (x_2, y_2) 到 (x_3, y_3)

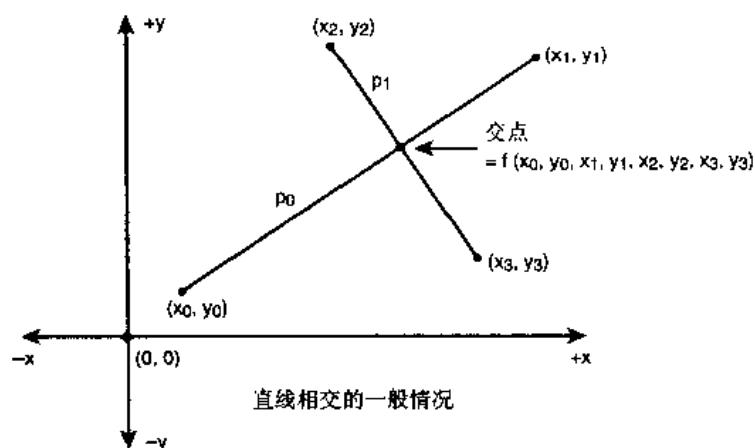


图 8.10 计算两直线的交点

这里 p_0 、 p_1 可以为任意方向。

式 1 为线 p_0 的点斜式：

$$m_0 = (y_1 - y_0) / (x_1 - x_0)$$

且有

$$(x - x_0) = m_0 (y - y_0)$$

式 2 为线 p_1 的点斜式：

$$m_1 = (y_3 - y_2) / (x_3 - x_2)$$

且有

$$(x - x_2) = m_1 (y - y_2)$$

如是你可以得到下列两式：

$$\text{式 1: } (x - x_0) = m_0 (y - y_0)$$

$$\text{式 2: } (x - x_2) = m_1 (y - y_2)$$

有两种办法可以解出 (x, y) 的值：迭代法或矩阵法。让我们先试一试迭代法。用一个公式中的变量表达出另外一个变量，然后代入另一个公式。让我们在式 1 中用 y 来表示 x ，然后代入式 2。

式 1 中 x 由 y 来表示：

$$(x - x_0) = m_0 (y - y_0)$$

$$x = m_0 (y - y_0) + x_0$$

很简单，让我们把 x 代入式 2：

$$\text{式 2 是 } (x - x_2) = m_1 (y - y_2)$$

在式 1 中， $x = m_0 (y - y_0) + x_0$ 。将它代入 x ：

$$(m_0 (y - y_0) + x_0 - x_2) = m_1 (y - y_2)$$

简化求 y :

$$m_0y - m_0y_0 + x_0 - x_2 = m_1y - m_1y_2$$

整理公式得:

$$m_0y - m_1y = -m_1y_2 - (-m_0y_0 + x_0 - x_2)$$

提取 y :

$$y(m_0 - m_1) = m_0y_0 - m_1y_2 + x_2 - x_0$$

最后, 两边同时除以 $(m_0 - m_1)$ 可以得出:

$$y = (m_0y_0 - m_1y_2 + x_2 - x_0) / (m_0 - m_1)$$

此时我们可以代回式 1 求出 x 值, 你也可以重新利用式 2 得到 x 的表达式取代式 1 中的 y , 结果都是一样的, 如下所示。

式 3:

$$x = -(m_0 / (m_1 - m_0)) x_2 + m_0 (y_2 - y_0) + x_0$$

式 4:

$$y = (m_0y_0 - m_1y_2 + x_2 - x_0) / (m_0 - m_1)$$

下面, 有几点你必须考虑。首先, 有没有使前面公式出问题的条件? 是的! 在现代数学中, 无穷并不会有什么麻烦, 但在计算机图形中却会。在式 3 和式 4 中, 如果两条直线斜率相同, 即两条直线平行, 项 $(m_1 - m_0)$ 或 $(m_0 - m_1)$ 就为 0。

这时, 直线不可能相交, 分母为 0, 式 3 和 4 的值为无穷大。当然, 这表明只有在无穷远处直线才可能相交, 但是你仅仅工作在一个 1024×768 之类分辨率的计算机屏幕上, 这个问题是无需考虑的。

这表明, 你的直线相交公式只有在直线相交时才起作用! 如果它们不能够相交, 数学就出了问题。不过, 这很好进行测试, 只要在进行数学计算之前, 进行一下 m_1 和 m_0 的比较, 如果相等, 就表明它们不相交, 总之让我们继续吧。

如果你仔细看看公式 3 和公式 4, 你就会发现, 你一共进行了 4 次除法运算, 4 次乘法运算, 8 次加法运算 (这里把减法运算也看成加法运算)。如果你还要计算斜率 m_0 和 m_1 , 另外还需要加上 4 次加法运算和两次除法运算, 还不算太糟糕。

采用普通表达式计算两条直线的交点

任何直线都可以表示为:

$$ax = by = c$$

或者

$$ax + by + c = 0$$

事实上, 两点式和截距式都可以转变为普通式, 比如下面的截距式。

$$y = mx + b$$

$$mx - ly = b$$

或者 $a=m$, $b=-1$, 且 $c=b$ (截距)。如果没有截距, 只有 (x_0, y_1) , (x_2, y_2) 两点的坐标怎么办? 好, 让我们来看看点斜式吧。

$$(y - y_0) = m(x - x_0)$$

乘以 m :

$$y - y_0 = mx - m x_0$$

将 x, y 移到公式的左边:

$$-mx + y = y_0 - m x_0$$

乘以 -1 :

$$mx + (-1)y = (-m) x_0 + (-1) y_0$$

数 学

-1 不一定需要, 只是为了看清楚 (a, b, c) 。

采用矩阵计算两条直线的交点

看来可能是 $a=m$, $b=-1$, $c=(-mx_0 - y_0)$ 。现在你知道了如何把点斜式转变成为普通式, 你也可以采用矩阵的解法。让我们来看看。

下面给出了两条直线的表达式:

$$a_1x + b_1y = c_1$$

$$a_2x + b_2y = c_2$$

现在想通过同时解两个方程得到 (x, y) 。在前面的例子中你使用了替代法, 这里还有基于矩阵另一种解法。现在我只是把原理给你讲清楚, 因为, 在后面 3D 中, 还要给你大量补充矢量/矩阵数学。从现在开始, 我将把结果和如何进行操作矩阵告诉你。看下面的例子。

让矩阵 A 等于:

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}$$

且变量 X 等于:

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

之后, 变量 Y 等于:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

因此, 你可以得出下面矩阵表达式:

$$A \times X = Y$$

两边同时乘以矩阵 A 的逆阵即 A^{-1} , 可以得出:

$$A^{-1} \times A \times X = A^{-1} \times Y$$

简化为:

$$X = A^{-1} \times Y$$

完成! 当然, 你必须知道如何求矩阵的逆阵, 并且知道矩阵的乘法运算并得出(x, y)。
我这里来给你一些帮助并给出其最后结果。

$$x = \text{Det}(A1) / \text{Det}$$

$$y = \text{Det}(A2) / \text{Det}$$

这里 A1 等于:

$$\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}$$

A2 等于:

$$\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}$$

总之, 你必须用 Y 分别取代 A 阵的第一、第二列来得到 A1、A2。Det(M)表示 M 值由下面方法来确定(一般情况下)。

给出一个普通的 2×2 矩阵 M:

$$M = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

$$\text{Det}(M) = (ad - cb)$$

记住上面这些, 下面给出一个例子。

$$A \times X = Y$$

$$5x - 2y = -1$$

$$2x + 3y = 3$$

$$A = \begin{vmatrix} 5 & -2 \\ 2 & 3 \end{vmatrix}$$

$$X = \begin{vmatrix} x \\ y \end{vmatrix}$$

$$Y = \begin{vmatrix} -1 \\ 3 \end{vmatrix}$$

因此:

$$A1 = \begin{vmatrix} -1 & -2 \\ 3 & 3 \end{vmatrix}$$

$$A2 = \begin{vmatrix} 5 & -1 \\ 2 & 3 \end{vmatrix}$$

解出 x, y 的值:

$$x = \frac{\text{Det} \begin{vmatrix} -1 & -2 \\ 3 & 3 \end{vmatrix}}{\text{Det} \begin{vmatrix} 5 & -1 \\ 2 & 3 \end{vmatrix}} = \frac{(-1 \times 3 - 3 \times (-2))}{(5 \times 3 - 2 \times (-2))} = 3/19$$

$$y = \frac{\text{Det} \begin{vmatrix} 5 & -1 \\ 2 & 3 \end{vmatrix}}{\text{Det} \begin{vmatrix} -1 & -2 \\ 3 & 3 \end{vmatrix}} = \frac{(5 \times 3 - 2 \times (-1))}{(-1 \times 3 - 3 \times (-2))} = 17/19$$

哇,看起来像在演戏,不是吗?好,游戏编程实际上都是数学编程,尤其是在今天。幸运的是,你写完数学代码后,无需再为它担心。但是理解它非常重要,所以我带你进行了一下温习。

学过数学之后,让我们回到主题上来——剪切。

剪切直线

如你所见,剪切的观念虽然简单,但是由于需要线性代数,实际操作并不简单。至少你应该知道如何处理线性公式并计算其如何相交,但是如我们前面所言,你总是可以利用关于这些问题的预备知识,来简化数学问题。

我们距用一个矩形剪切直线仍有一段距离,但我们最终会做到这一点。现在,让我们来看一看问题所在,看看用一条垂直线和一条水平线对一条普通直线的剪切是否有用。下面看看图 8.11。

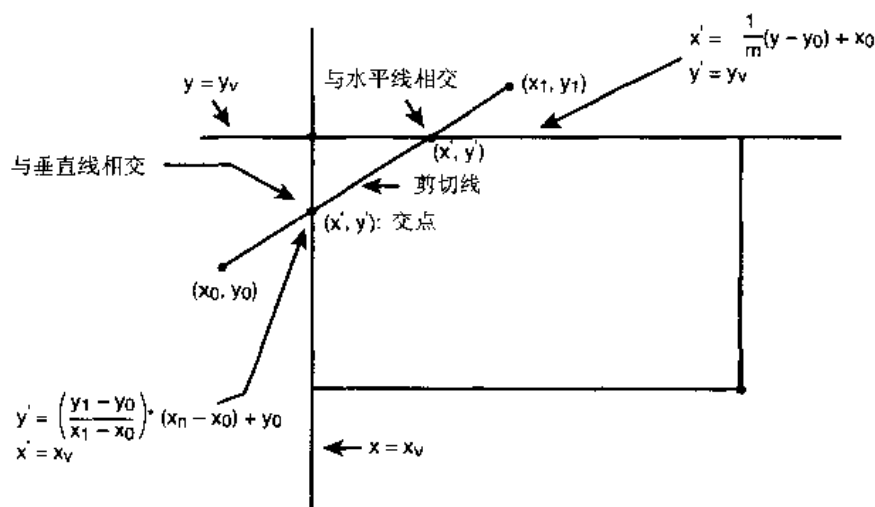


图 8.11 矩形剪切比一般情况都简单

从图 8.11 可以看出，你一次仅仅需要考虑一个变量，或者 x ，或者 y 。这不过是很简单的数学。基本上，无需经过很复杂的数学运算(在 3D 中往往需要考虑)，你只是需要将式 $X=\text{常量}$ ， $Y=\text{常量}$ ，直接代入直线的点斜式，就可以得到所要的值。例如，我们需要剪切的区域是 (x_1, y_1) ， (x_2, y_2) 。如果已经知道了左侧交点的值的 x 坐标的值为 x_1 ，想知道左边剪切点的值，只需要找出 y 坐标就可以了。

相反，如果你想找出水平线如剪切区底线上的交点，这里为 y_2 ，你已经知道了 y 坐标为 y_2 ，便只需找出 X ，不是吗？这里是通过数学计算得出直线 (x_0, y_0) 到 (x_1, y_1) 同水平直线 $Y=Y_h$ 和垂直直线 $X=X_v$ 的交点 (x, y) 。

水平线的交点 (x, Y_h) :

求 x

从点斜式开始: $m=(y_1-y_0)/(x_1-x_0)$

$$(y-y_0) = m(x-x_0)$$

$$(y-y_0) = mx-mx_0$$

$$(y-y_0)+mx_0 = mx$$

$$((y-y_0)+mx_0)=x$$

$$x=((y-y_0)+mx_0)$$

或者

$$x = \frac{1}{m}(y-y_0)+x_0$$

垂直线交点(X_v, y)

求 y

从点斜式开始: $m = (y_1 - y_0) / (x_1 - x_0)$

$(y - y_0) = m(x - x_0)$

$y = m(x - x_0) + y_0$

这就是求交点的步骤, 所以, 现在你可以计算任意一条直线同一条水平或垂直线(矩形剪切区中最关键的部分)的交点。因此, 我们可以进行剪切区其余问题的讨论。

Cohen-Sutherland 算法

一般情况下, 你需要决定一条直线是全部可见、部分可见、部分剪切(一端剪切)或者是全部剪切(两端剪切)。这必须进行处理, 已经发明了许多算法处理所有情况, 其中一种算法得到了最为广泛的应用: Cohen-Sutherland 算法。它速度既快, 又不是太复杂, 因此得到了广泛的使用。

基本上, 它是一个野蛮的算法。但是, 它并没有用成百万句“if 语句”来寻找直线到底在什么地方, 而是将剪切区分成许多部分, 然后将每一段被剪切的线段的一端分配一位代码, 而后仅仅采用一些“if 语句”或一个“case”语句, 判断出情况到底如何, 图 8.12 给出了原理示意图。

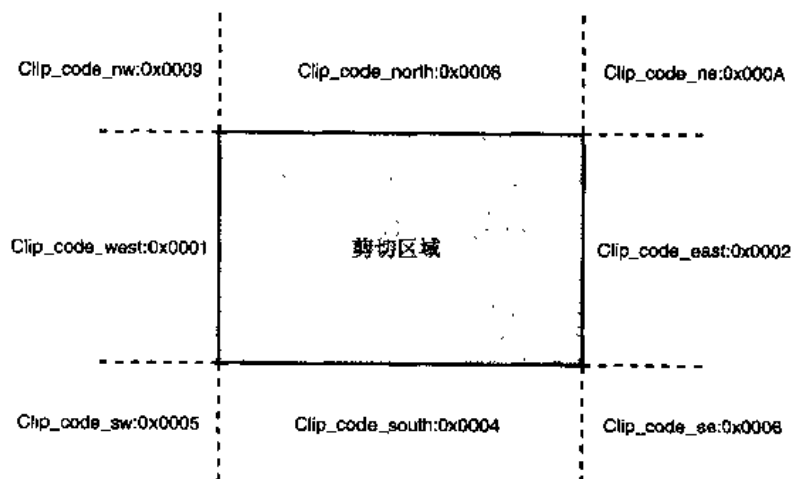


图 8.12 用剪切代码进行有效线段的端点检测

下面的函数是我根据同样的原理编写的 Cohn-Sutherland 算法的一个版本。

```
int Clip_Line(int &x1,int &y1,int &x2, int &y2)
{
// this function clips the sent line using the globally defined clipping
// region
```

```
// internal clipping codes
#define CLIP_CODE_C 0x0000
#define CLIP_CODE_N 0x0008
#define CLIP_CODE_S 0x0004
#define CLIP_CODE_E 0x0002
#define CLIP_CODE_W 0x0001

#define CLIP_CODE_NE 0x000a
#define CLIP_CODE_SE 0x0006
#define CLIP_CODE_NW 0x0009
#define CLIP_CODE_SW 0x0005

int xc1=x1,
    yc1=y1,
    xc2=x2,
    yc2=y2;

int p1_code=0,
    p2_code=0;

// determine codes for p1 and p2
if (y1 < min_clip_y)
    p1_code|=CLIP_CODE_N;
else
if (y1 > max_clip_y)
    p1_code|=CLIP_CODE_S;

if (x1 < min_clip_x)
    p1_code|=CLIP_CODE_W;
else
if (x1 > max_clip_x)
    p1_code|=CLIP_CODE_E;

if (y2 < min_clip_y)
    p2_code|=CLIP_CODE_N;
else
if (y2 > max_clip_y)
    p2_code|=CLIP_CODE_S;

if (x2 < min_clip_x)
    p2_code|=CLIP_CODE_W;
else
if (x2 > max_clip_x)
    p2_code|=CLIP_CODE_E;

// try and trivially reject
if ((p1_code & p2_code))
    return(0);

// test for totally visible, if so leave points untouched
```

```

if (p1_code==0 && p2_code==0)
    return(1);

// determine end clip point for p1
switch(p1_code)
{
    case CLIP_CODE_C: break;

    case CLIP_CODE_N:
    {
        yc1 = min_clip_y;
        xc1 = x1 + 0.5*(min_clip_y-y1)*(x2-x1)/(y2-y1);
    } break;

    case CLIP_CODE_S:
    {
        yc1 = max_clip_y;
        xc1 = x1 + 0.5*(max_clip_y-y1)*(x2-x1)/(y2-y1);
    } break;

    case CLIP_CODE_W:
    {
        xc1 = min_clip_x;
        yc1 = y1 + 0.5*(min_clip_x-x1)*(y2-y1)/(x2-x1);
    } break;

    case CLIP_CODE_E:
    {
        xc1 = max_clip_x;
        yc1 = y1 + 0.5*(max_clip_x-x1)*(y2-y1)/(x2-x1);
    } break;

    // these cases are more complex, must compute 2 intersections
    case CLIP_CODE_NE:
    {
        // north hline intersection
        yc1 = min_clip_y;
        xc1 = x1 + 0.5*(min_clip_y-y1)*(x2-x1)/(y2-y1);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xc1 < min_clip_x || xc1 > max_clip_x)
        {
            // east vline intersection
            xc1 = max_clip_x;
            yc1 = y1 + 0.5*(max_clip_x-x1)*(y2-y1)/(x2-x1);
        } // end if

    } break;

    case CLIP_CODE_SE:
    {

```

```

        // south hline intersection
        ycl = max_clip_y;
        xcl = x1 + 0.5*(max_clip_y-y1)*(x2-x1)/(y2-y1);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xcl < min_clip_x || xcl > max_clip_x)
        {
            // east vline intersection
            xcl = max_clip_x;
            ycl = y1 + 0.5*(max_clip_x-x1)*(y2-y1)/(x2-x1);
        } // end if

    } break;

case CLIP_CODE_NW:
    {
        // north hline intersection
        ycl = min_clip_y;
        xcl = x1 + 0.5*(min_clip_y-y1)*(x2-x1)/(y2-y1);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xcl < min_clip_x || xcl > max_clip_x)
        {
            xcl = min_clip_x;
            ycl = y1 + 0.5*(min_clip_x-x1)*(y2-y1)/(x2-x1);
        } // end if

    } break;

case CLIP_CODE_SW:
    {
        // south hline intersection
        ycl = max_clip_y;
        xcl = x1 + 0.5*(max_clip_y-y1)*(x2-x1)/(y2-y1);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xcl < min_clip_x || xcl > max_clip_x)
        {
            xcl = min_clip_x;
            ycl = y1 + 0.5*(min_clip_x-x1)*(y2-y1)/(x2-x1);
        } // end if

    } break;

default:break;

} // end switch

```



```

// determine clip point for p2
switch(p2_code)
{
    case CLIP_CODE_C: break;

    case CLIP_CODE_N:
    {
        yc2 = min_clip_y;
        xc2 = x2 + (min_clip_y-y2)*(x1-x2)/(y1-y2);
    } break;

    case CLIP_CODE_S:
    {
        yc2 = max_clip_y;
        xc2 = x2 + (max_clip_y-y2)*(x1-x2)/(y1-y2);
    } break;

    case CLIP_CODE_W:
    {
        xc2 = min_clip_x;
        yc2 = y2 + (min_clip_x-x2)*(y1-y2)/(x1-x2);
    } break;

    case CLIP_CODE_E:
    {
        xc2 = max_clip_x;
        yc2 = y2 + (max_clip_x-x2)*(y1-y2)/(x1-x2);
    } break;

    // these cases are more complex, must compute 2 intersections
    case CLIP_CODE_NE:
    {
        // north hline intersection
        yc2 = min_clip_y;
        xc2 = x2 + 0.5*(min_clip_y-y2)*(x1-x2)/(y1-y2);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xc2 < min_clip_x || xc2 > max_clip_x)
        {
            // east vline intersection
            xc2 = max_clip_x;
            yc2 = y2 + 0.5*(max_clip_x-x2)*(y1-y2)/(x1-x2);
        } // end if

    } break;

    case CLIP_CODE_SE:
    {
        // south hline intersection
        yc2 = max_clip_y;

```

```

        xc2 = x2 + 0.5+(max_clip_y-y2)*(x1-x2)/(y1-y2);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xc2 < min_clip_x || xc2 > max_clip_x)
        {
            // east vline intersection
            xc2 = max_clip_x;
            yc2 = y2 + 0.5+(max_clip_x-x2)*(y1-y2)/(x1-x2);
        } // end if

    } break;

case CLIP_CODE_NW:
    {
        // north hline intersection
        yc2 = min_clip_y;
        xc2 = x2 + 0.5+(min_clip_y-y2)*(x1-x2)/(y1-y2);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xc2 < min_clip_x || xc2 > max_clip_x)
        {
            xc2 = min_clip_x;
            yc2 = y2 + 0.5+(min_clip_x-x2)*(y1-y2)/(x1-x2);
        } // end if

    } break;

case CLIP_CODE_SW:
    {
        // south hline intersection
        yc2 = max_clip_y;
        xc2 = x2 + 0.5+(max_clip_y-y2)*(x1-x2)/(y1-y2);

        // test if intersection is valid,
        // if so then done, else compute next
        if (xc2 < min_clip_x || xc2 > max_clip_x)
        {
            xc2 = min_clip_x;
            yc2 = y2 + 0.5+(min_clip_x-x2)*(y1-y2)/(x1-x2);
        } // end if

    } break;

default:break;

} // end switch

// do bounds check
if ((xc1 < min_clip_x) || (xc1 > max_clip_x) ||

```

```

    (yc1 < min_clip_y) || (yc1 > max_clip_y) ||
    (xc2 < min_clip_x) || (xc2 > max_clip_x) ||
    (yc2 < min_clip_y) || (yc2 > max_clip_y) )
    {
        return(0);
    } // end if

    // store vars back
    x1 = xc1;
    y1 = yc1;
    x2 = xc2;
    y2 = yc2;

    return(1);

} // end Clip_Line

```

现在，你需要做的就是将直线的端点发送给函数，它将用由全局变量定义的剪切区进行剪切。

```

int min_clip_x = 0, // clipping rectangle
    max_clip_x = SCREEN_WIDTH-1,
    min_clip_y = 0,
    max_clip_y = SCREEN_HEIGHT-1;

```

我通常将剪切区设为屏幕的尺寸。函数的关键在于用引用定义参数，所以变量是可以更改的。如果你不想改变变量，复制一下即可。下面是函数应用的实例。

```

// clip the line (x1, y1) to (x2, y2)

// make copies
int clipped_x1 = x1,
    clipped_y1 = y1,
    clipped_x2 = x2,
    clipped_y2 = y2;

// clip the line
Clip_Line(clipped_x1, clipped_y1,
          clipped_x2, clipped_y2);

```

当函数返回“clipped_*”变量后，它们又被储存在全局变量中的矩形剪切区赋予新的剪切值。

自演示程序 DEMO8_2.CPP1EXE，在屏幕中心创建了一个 200×200 的剪切区，然后在其中画随机线。注意它们的剪切是如何进行的。

线框多边形

既然你已经能够画直线并且通过矩形剪切框对它们进行剪切，现在就可以进入更高的议题——多边形上来了。图 8.13 给出了几种多边形：三角形、四边形、五角形。多边形有三个或更多个节点且线路闭合。而且多边形既可以是凸多边形，也可以是凹多边形。一般认为，凸多边形没有“齿”，而凹多边形却有。

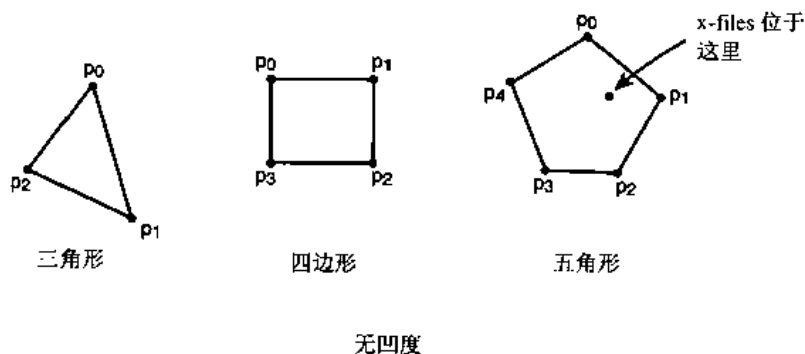


图 8.13 一般的多边形

数 学

一种检验多边形凹凸的方法是：如果能从多边形任意两边取点作为端点画线，而使该直线落在多边形的外面，则多边形为凹。

现在，我想给你一些如何取代及操纵 2D 多边形对象的建议。

多边形的数据结构

在游戏中，数据结构的选择至关重要，把你所学的所有数据结构忘掉，仅仅记住一件事情——速度。在游戏中，你不得不随时利用数据，因为游戏的着色依赖于它。你必须考虑数据使用的方便性，数据的大小，处理器缓冲访问的数据的相对大小，甚至考虑二级缓冲或者更深一级的缓冲。结果是即使你有一个 1000MHz 的处理器，但是没有快速、高效的获得数据的方法也是白搭。

我的数据结构设计原则是：

- 简洁。
- 已知大小的小的数据结构在 25% 的范围内采用静态数组。
- 需要时采用数据链。

- 不要仅仅因为“酷”，而是当能够使代码运行快时才使用树形或奇异数据结构。
- 最后，当进行数据结构设计时深思熟虑，不要认为自己不能够对设计对象进行创新和发明。

总之，说得够多了。让我们来看一个画多边形的基本数据结构吧。假设多边形有很多个顶点，这就限制了采用静态数组来存储顶点，这时应该采用动态数组来存储具体的顶点。不但如此，你还需要多边形的位置(x, y)、速度(后面会看到)、颜色等其他无法想像的可能状态信息。先看下面的例子：

```
typedef struct POLYGON2D_TYP
{
    int state;           // state of polygon
    int num_verts;       // number of vertices
    int x0, y0;          // position of center of polygon
    int xv, yv;          // initial velocity
    DWORD color;         // could be index or PASLETTEENTRY
    VERTEX2DI *vlist     // pointer to vertex list
} POLYGON2D, *POLYGON2D_PTR;
```

C++

也可以采用 C++ 和类，这里为了简单，假设你是一个 C 用户。但是，作为练习，我希望所有的 C++ 程序员把多边形的数据结构转换成类。

到现在为止，你还没有定义 VERTEX2DI。对你而言，这又是一个进行数据结构设计的典型。你不必定义所有的东西，但是必须知道你需要什么东西，现在我们开始 VERTEX2DI 的定义。基本上，它仅仅是一个精确到整数位的 2D 矢量。

```
typedef struct VERTEX2DI_TYP
{
    int x, y;
} VERTEX2DI, *VERTEX2DI_PTR
```

数 学

在许多 2D/3D 的引擎中，所有的矢量对整数来说是精确的，当然，这时的放大或者旋转变得不是那么精确。浮点数的问题在于它们转变成整数的速度较慢。即使奔腾处理器处理浮点数的速度和处理整数的速度一样快或者更快，最后进行的整数光栅化变换也将置你于死地。最终的转换并不是问题，问题是如果你必须来回反复，就将最终毁了你游戏的表现。结论是，如果整数精度能够满足，就绝对要这样做。另外，一旦采用浮点数，就放弃转换。

至此，你有了好的数据结构保存矢量和多边形，图 8.14 给出了真正的多边形同数据结构关系的示意图。

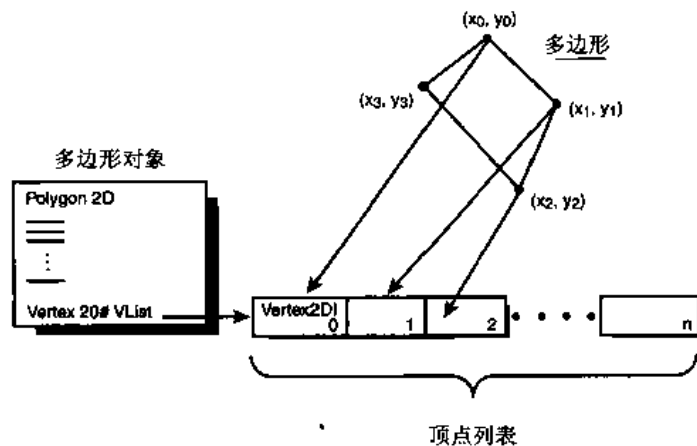


图 8.14 多边形数据结构

对于你所使用的多边形数据结构，需要你做的一件事是为实际的矢量存储分配内存，你需要做以下这些事：

```
POLYGON2D triangle;          // our polygon

// initialize the triangle
triangle.state      =1;       // turn it on
triangle.num_verts  =3;       // triangle
triangle.x0         =100;     // position it
triangle.y0         =100;
triangle.xv         =0;       // initial velocity
triangle.yv         =0;
triangle.color       =50;     // assume 8-bit mode index 50
triangle.vlist       =new VERTEX2DI[triangle.num_verts];
```

C++

注意到我采用了 C++ 语言的“new”操作分配了内存，因此，还将不得不用“delete”操作进行删除，在 C 语言中，用类似下面的语句分配内存。

```
(VERTEX2DI_PTR)malloc(triangle.num_verts*sizeof(VERTEX2DI))
```

妙极了！下面看我们如何来绘制它们。

多边形的绘制及剪切

画多边形和绘制 n 条相连的直线段一样简单，你需要做的仅仅是节点的循环和连接。当然，如果想对多边形进行剪切，还需要调用剪切函数。此函数我已经封装在 Draw_Clip_line() 函数中了。函数具有同 Draw_Line() 函数一样的参数，不过，它剪切的区域是全局变量定义的区域。下面是一个画多边形数据结构的普通函数。

```
int Draw_Polygon2D(POLYGON2D_PTR poly, UCHAR *vbuffer, int lpitch)
{
```

```

// this function draws a POLYGON2D based on

// test if the polygon is visible
if (poly->state)
{
    // loop thru and draw a line from vertices 1 to n
    for (int index=0; index < poly->num_verts-1; index++)
    {
        // draw line from ith to ith+1 vertex
        Draw_Clip_Line(poly->vlist[index].x+poly->x0,
                        poly->vlist[index].y+poly->y0,
                        poly->vlist[index+1].x+poly->x0,
                        poly->vlist[index+1].y+poly->y0,
                        poly->color,
                        vbuffer, lpitch);

        } // end for

    // now close up polygon
    // draw line from last vertex to 0th
    Draw_Clip_Line(poly->vlist[0].x+poly->x0,
                    poly->vlist[0].y+poly->y0,
                    poly->vlist[index].x+poly->x0,
                    poly->vlist[index].y+poly->y0,
                    poly->color,
                    vbuffer, lpitch);

    // return success
    return(1);
} // end if
else
    return(0);

} // end Draw_Polygon2D

```

惟一不可思议的事是函数在画多边形时采用了 (x_0, y_0) 中心坐标。这使得你可以在周围移动多边形而不至于丢失单个的顶点。另外，采用相对于中心点坐标的方式可以使你利用本地坐标系，而不是世界坐标系，从图 8.15 中可以看出两者之间的关系。

采用相对于中心点 $(0, 0)$ (本地坐标系)然后再转变成点 (x, y) (世界坐标系)非常好。你将在 3D 中接触到具体的本地、世界、摄影(同视点有关)坐标系，现在只要知道它们的存在就可以了。作为一个演示程序，我编写了 DEMO8_3.CPPIEXE。程序先是创建一个多边形阵，每个多边形有八个节点。多边形看起来像是星形线。然后程序随机改变它们的位置，让它们在屏幕上到处移动。程序采用 $640 \times 480 \times 8$ 的工作方式，并采用页交换技术使程序动起来。图 8.16 给出了程序运行中的一个画面。

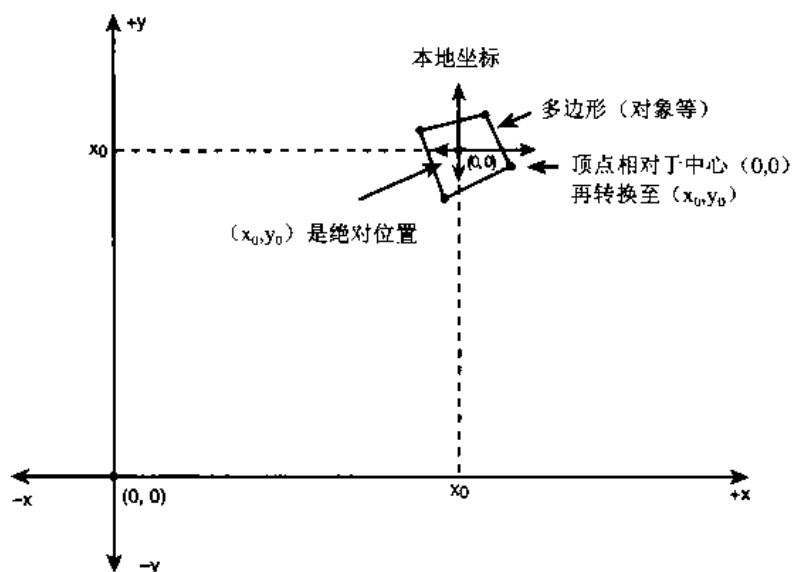


图 8.15 本地坐标和世界坐标的关系

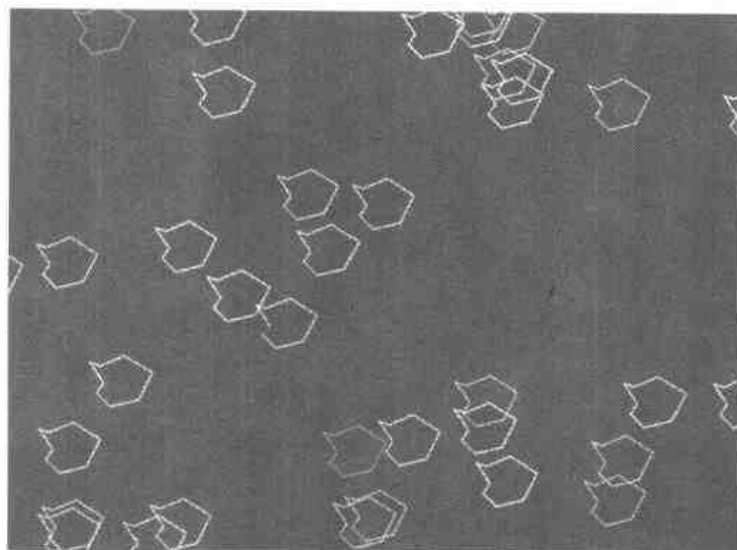


图 8.16 运行 DEM08_3.EXE

好了，现在你可以定义一个多边形并画出它来了。下一个议题将是 2D 图形的变换——平移、旋转和缩放。

2D 平面的变换

我想你肯定会发现：无形中我已将你带入了数学世界之中。然而，如果你理解了基本原理，你就会发现很有趣，而且 3D 游戏的编程也是如此，没有什么大不了的，现在就让

我们开始吧。

到此为止，你已经很多次听到了变换这个词，但是你还仍然没有接触到它的数学描述，包括移动和旋转。让我们来看看这些基本概念，并看看它们是如何同 2D 矢量图形联系在一起的。而后，当你到了 3D 图形部分，你只要加入一两个变量并考虑一下 Z 轴就可以了。

平移

平移无非是对象或点从一个地方移到另一个地方。假设你有一个点 (x, y) ，你想将它平移一定距离 (dx, dy) 。移动过程的示意图如图 8.17 所示。

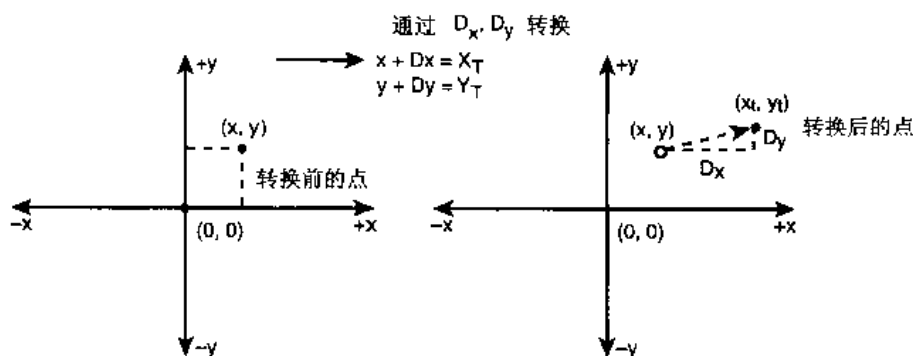


图 8.17 移动一个单点

基本上，你对 (x, y) 点加上了一个移动因子，到了 (x_1, y_1) 的新位置。下面是其数学过程：

$$x_1 = x + dx;$$

$$y_1 = y + dy;$$

这里， dx, dy 可正可负。如果我们采用标准的显示器坐标， $(0, 0)$ 为左上角，那么，正的 x 移动量使对象向右平移，正的 y 移动量使对象向下平移，负的 x 移动量向左，负的 y 移动量向上。

平移整个对象时，如果对象有中心，且其他所有点(如多边形结构)都相对于该中心，你只需移动该中心点 (x, y) 。如果没有中心点 (x, y) ，则你必须将此式应用于组成多边形的所有点，这使我们想起了本地坐标和绝对坐标的概念。

通常，你要定义的 2D/3D 计算机图形至少要有本地坐标或世界坐标。一个对象的本地坐标是相对于 $(0, 0)$ 而言，在 3D 中相对于 $(0, 0, 0)$ 。如果平移 (x_0, y_0) 使本地坐标到对象上，可以通过将 (x_0, y_0) 加到每个本地坐标上得到世界坐标系。这如图 8.18 所示。

考虑到这点，将来你可以考虑在多边形上加上一些存储数据，使你能够存储本地坐标和世界坐标。实际上，你将在后面做这些工作。另外，你还需要添加摄影坐标的存储。增加存储的原因是：一旦你将一个对象变换成世界坐标并要画出它，而又不想在每个框中都

做这件事时，只要对象没有移动或变换，你就可以不做。你只要存储最后计算出的世界坐标就可以了。

记住这些，让我们来看一个通用的多边形移动函数。它很简单，应该没有什么错误。

```
int Translate_Polygon2D(POLYGON2D_PTR poly, int dx, int dy)
{
    // this function translates the center of a polygon

    // test for valid pointer
    if (!poly)
        return(0);

    // translate
    poly->x0+=dx;
    poly->y0+=dy;

    // return success
    return(1);

} // end Translate_Polygon2D
```

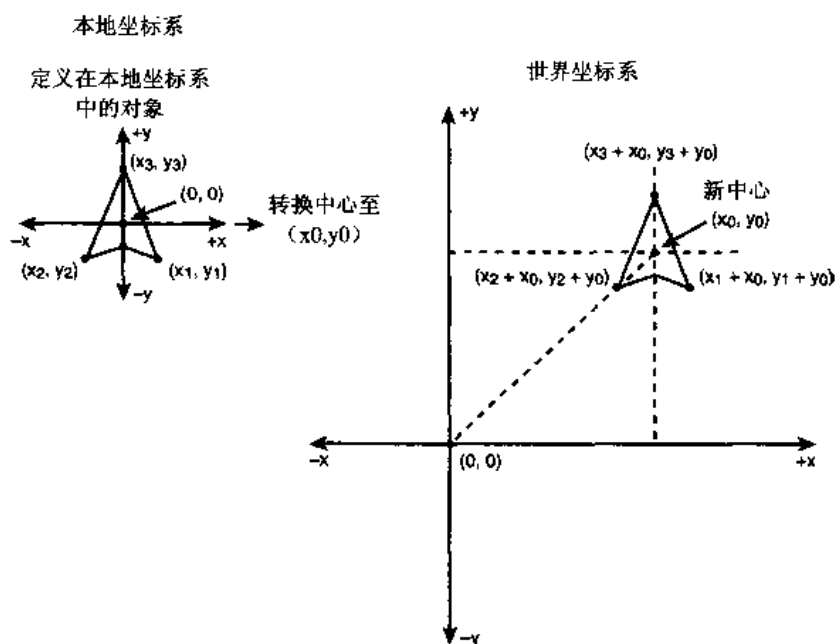


图 8.18 物体移动及矢量结果

旋转

位图旋转十分复杂，而在平面中的单点旋转则问题不大，至少实际旋转如此。产生旋转有点复杂，不过我有一个很酷的产生旋转的方法。但是在介绍它之前，让我们来看一下

我们到底想干什么。参考图 8.19，你看到点 p_0 的坐标为 (x, y) ，你想将该点绕着 z 轴（穿过纸面）旋转一个角度到 p_0' ，现在求旋转坐标 (x_r, y_r) 。

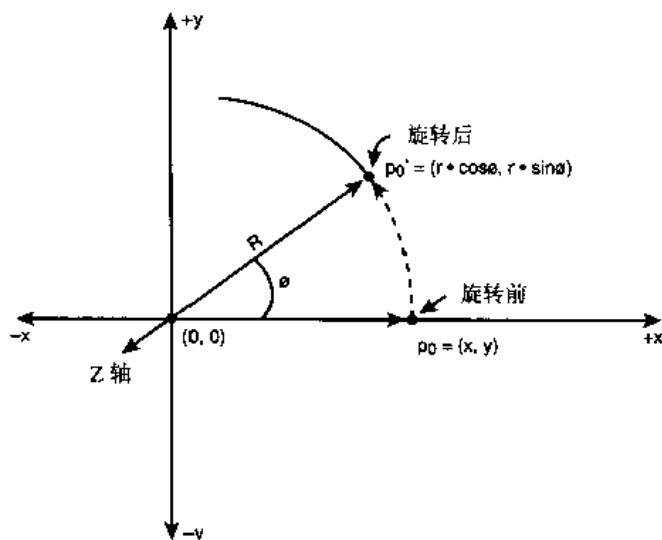


图 8.19 点的旋转

三角函数回顾

很明显，这里将用到三角形的知识，你可能有点忘记了，现在让我们对一些基本点进行一下回顾。

多数三角学概念可用如图 8.20 所示三角形表示。

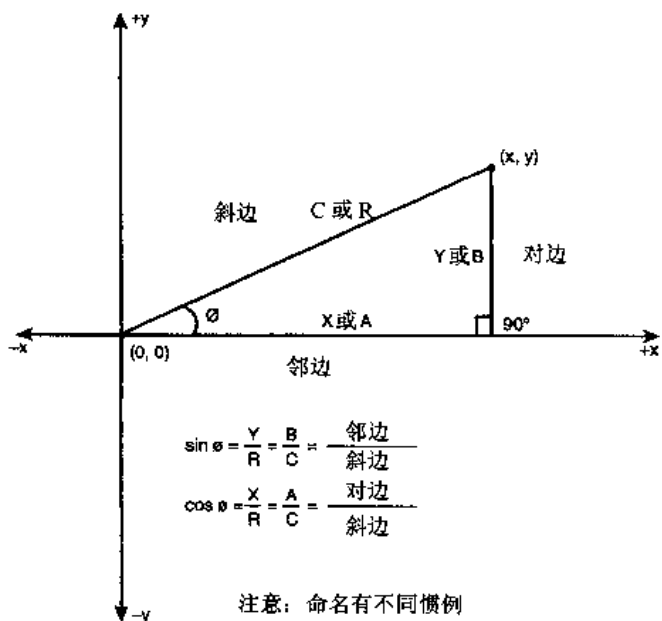


图 8.20 直角三角形

表 8-1 给出了弧度和角度的差异。

表 8.1 弧度和角度的差异

角 度	弧度 (π)	弧度 (数字表示)
360	2π	6.28
180	π	3.14159
90	$\pi/2$	1.57
57.295	π/π	1.0
1	$\pi/180$	0.0175

下面是一些三角学基本概念。

- 360 度，也就是 2π ，为一周。因此 π 为 180 度。记住，计算机函数 $\sin()$ 、 $\cos()$ 采用的是弧度，而不是角度。表 8.1 列出了这些值。
- 三角形的内角之和为 180 度或 π 。
- 参见图 8.20 中的直角三角形， θ 角对着的边称为对边，相邻的边称为邻边，最长的一边称为斜边。
- 两条直角边的平方和等于斜边的平方。这被称为勾股定理（毕达哥拉斯定理）。可以用数学表达为：

$$\text{斜边}^2 = \text{对边}^2 + \text{邻边}^2$$

或者，使用 a 、 b 、 c 来作为变量：

$$c^2 = a^2 + b^2$$

因此，如果有了一个三角形的两条边，也可以得到第三条边。

- 数学家喜欢使用 \sin 、 \cos 和 \tan 来表示三个主要的三角比率，分别定义为：

$$\cos \theta = \frac{\text{邻边}}{\text{斜边}} = \frac{x}{r}$$

定义域： $0 \leq \theta \leq 2\pi$

值域：-1 到 1

$$\sin \theta = \frac{\text{对边}}{\text{斜边}} = \frac{y}{r}$$

定义域： $0 \leq \theta \leq 2\pi$

值域：-1 到 1

$$\tan \theta = \frac{\sin \theta}{\cos \theta} = \frac{\text{对边/斜边}}{\text{邻边/斜边}} = \frac{\text{对边}}{\text{邻边}} = \frac{y}{x} = \text{斜率} = M$$

定义域： $-\pi/2 \leq \theta \leq \pi/2$

值域： $-\infty$ 到 ∞

数 学

应当注意定义域和值域名词的使用。定义域和值域分别表示输入和输出。

图 8.21 表示了所有函数的图形。注意几个函数都是周期性的（重复出现）， $\sin \theta$ 和 $\cos \theta$ 函数的周期为 2π ，而 $\tan \theta$ 函数周期为 π 。注意当 θ 趋近于 $\pi/2$ 时， $\tan \theta$ 函数趋近于无穷大。

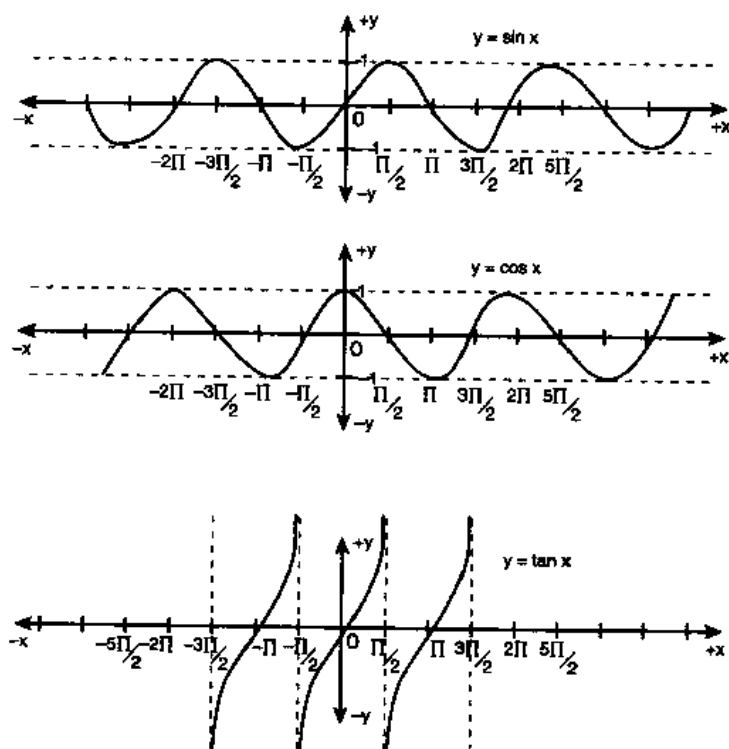


图 8.21 基本三角函数的图形

如果对三角恒等关系不清楚，请阅读数学书。我只准备介绍作为游戏程序员应当知道的一些基本内容而已。表 8.2 列出了一些三角函数及其关系公式。

表 8.2 常用的三角函数恒等式

Cosecant	$\csc \theta = 1/\sin \theta$
Secant	$\sec \theta = 1/\cos \theta$
Cotangent	$\cot \theta = 1/\tan \theta$

三角函数的毕达哥拉斯定理：

$$\sin^2 \theta + \cos^2 \theta = 1$$

变换恒等式：

$$\sin \theta = \cos(\theta - \frac{\pi}{2})$$

反射定律:

$$\sin(-\theta) = -\sin(\theta)$$

$$\cos(-\theta) = \cos(\theta)$$

加法定律:

$$\sin(\theta_1 + \theta_2) = \sin \theta_1 \cdot \cos \theta_2 + \cos \theta_1 \cdot \sin \theta_2$$

$$\cos(\theta_1 + \theta_2) = \cos \theta_1 \cdot \cos \theta_2 - \sin \theta_1 \cdot \sin \theta_2$$

$$\sin(\theta_1 - \theta_2) = \sin \theta_1 \cdot \cos \theta_2 - \cos \theta_1 \cdot \sin \theta_2$$

$$\cos(\theta_1 - \theta_2) = \cos \theta_1 \cdot \cos \theta_2 + \sin \theta_1 \cdot \sin \theta_2$$

当然, 还可以推导出更多的恒等式来。通常来说, 使用这些恒等式可以简化复杂的三角公式, 从而不必去推导数学公式。因此在编程过程中当提出一个基于 \sin 、 \cos 、 \tan 等等三角关系的算法时, 应当翻阅一下三角关系的书籍, 看是否能够简化数学计算, 以便减少计算结果的花费的时间。请记住: 速度、速度、还是速度! 速度是最重要的。

2D 平面的旋转

既然已经知道了什么是 \sin 、 \cos 、 \tan , 那么, 让我们来利用它们进行 2D 平面中的点的旋转。图 8.22 给出了旋转公式建立的示意图。

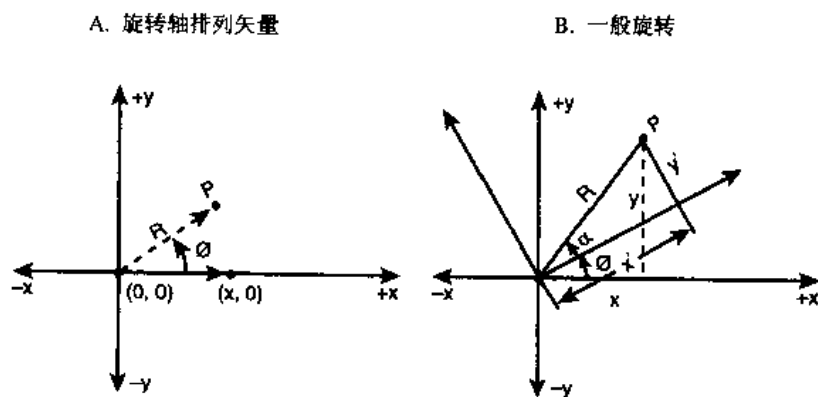


图 8.22 旋转公式的产生

现在从半径为 R 上的任意一点的计算开始。

$$xr = r \cdot \cos \theta$$

$$yr = r \cdot \sin \theta$$

因此, 如果你想旋转坐标为 $(x, 0)$ 的一个点, 你可以采用这个公式, 但是, 你还需要进行一般化。如果你想将任意一点 (x, y) 旋转一个 θ 角 (如图 8.22)。可以从两个方面考虑: 假

设点 P 旋转，或者假设轴本身旋转。如果认为是轴本身在旋转，那么需要有两套坐标：旋转前坐标和旋转后坐标。

在旋转前的坐标系中，你有：

$$xr = r \cdot \cos \theta$$

$$yr = r \cdot \sin \theta$$

旋转之后，你有

式一：

$$xr = r \cdot \cos(\theta + \alpha)$$

$$yr = r \cdot \sin(\theta + \alpha)$$

式二：

$$x = r \cdot \cos \alpha$$

$$y = r \cdot \sin \alpha$$

这里， α 是在旋转后，新的坐标系的 x 轴同从原点指向 P 点的矢量的夹角。

如果你对此感到迷惑，然我们从另一个角度来解释，你很容易找出旋转 θ 角度的点(x, 0)，如果你将轴旋转 θ 角，你就可以在新旧坐标系中算出 P 点。然后，基于这两个公式，你可以得到旋转公式。如果把式 1 用加法公式进行分解，可以得到

式 3：

$$xr = r \cdot \cos \theta \cdot \sin \alpha - r \cdot \sin \theta \cdot \sin \alpha$$

$$yr = r \cdot \sin \theta \cdot \cos \alpha - r \cdot \sin \theta \cdot \cos \alpha$$

等一下，让我来将 x, y 代入。你知道，x, y 也等于：

$$x = r \cdot \cos \alpha$$

$$y = r \cdot \sin \alpha$$

将它们代入式 3 中的黑体字部分就可以得出你想要的值：

式 4 旋转公式：

$$xr = r \cdot \cos \theta - y \cdot \sin \theta$$

$$yr = r \cdot \sin \theta + y \cdot \cos \theta$$

数 学

如果你有数学头脑，你就会发现结果很像极坐标向笛卡尔坐标的转变。这也是我推理它的原因。

回到具体中来，你现在知道可以通过 4 式将点(x, y)旋转 θ 角。但是，应该详细记住下面这一点：旋转时，逆时针旋转， θ 角为正，顺时针旋转， θ 为负。但是现在还有另外一个问题，公式是你在笛卡尔坐标系中的第一象限推出的，因此，在显示屏上，y 轴是相反方向，因此正负也跟着相反。

最后，当你处理 3D 图时，你需要对所有显示屏坐标进行变换，使 x, y 轴在中心，并且指向正方向，就像在 2D 笛卡尔坐标中的第一象限中一样。但是，目前无需费心。

多边形的旋转

发挥你所有的知识，让我们来写一个旋转多边形的函数。

```
int Rotate_Polygon2D(POLYGON2D_PTR poly, float theta)
{
    // this function rotates the local coordinates of the polygon

    // test for valid pointer
    if (!poly)
        return(0);

    // loop and rotate each point, very crude, no lookup!!!
    for (int curr_vert = 0; curr_vert < poly->num_verts; curr_vert++)
    {
        // perform rotation
        float xr =poly->vlist[curr_vert].x*cos(theta) -
                poly->vlist[curr_vert].y*sin(theta);

        float yr =poly->vlist[curr_vert].x*sin(theta) -
                poly->vlist[curr_vert].y*cos(theta);

        // store result back
        poly->vlist[curr_vert].x=xr;
        poly->vlist[curr_vert].y=yr;

    } // end for curr_vert

    // return success
    return(1);

} // end Rotate_Polygon2D
```

这里应该注意几点。首先数学运算采用的是浮点数，而保存的结果却是整数，因此损失了精度。

另外，函数使用的是弧度而不是角度，因为函数使用的库函数 `sin()`、`cos()` 就是采用弧度。精度的损失不是大问题，但是在实时程序中使用三角函数要多坏有多坏。你需要做的是创建一个查询表，将算好的 0~360 度的 `sin()`、`cos()` 值存储起来，然后由查询代替函数计算。

问题是如何进行表的设计？这要根据具体情况来定，有的程序想用一字节进行查询，那么，将一周分成 256 份就行了。如图 8.23 所示。

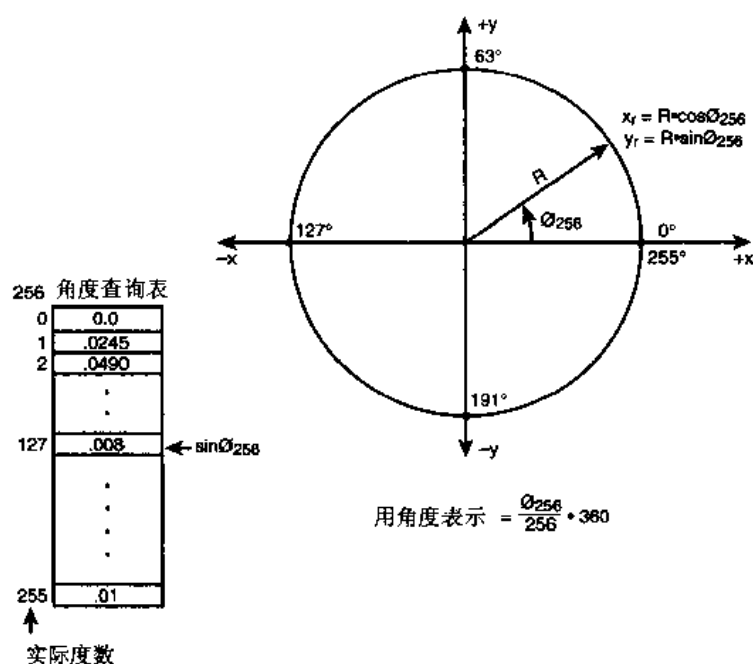


图 8.23 将圆分为 256 度

这要根据你的意愿和实际情况来定，但是，我通常喜欢将表分成 0~359 度，这里你可以这样创建：

```
//storage for our tables
float cos_look[360];
float sin_look[360];

// generate the tables
for (int ang =0; ang<360; ang++)
{
    // convert ang to radians
    float theta = (float)ang*3.14159/180;

    // invert next entry into table
    cos_look[ang] = cos(theta);
    sin_look[ang] = sin(theta);

} // end for ang
```

然后，我们将前面的程序再写一遍，用 sin_look[]、cos_look[] 数组分别代替 sin()、cos() 函数。

```
int Rotate_Polygon2D(POLYGON2D_PTR poly, int theta)
{
    // this function rotates the local coordinates of the polygon
```

```

// test for valid pointer
if (!poly)
    return(0);

// loop and rotate each point, very crude, no lookup!!!
for (int curr_vert = 0; curr_vert < poly->num_verts; curr_vert++)
{
    // perform rotation
    float xr = poly->vlist[curr_vert].x*cos_look[theta] -
              poly->vlist[curr_vert].y*sin_look[theta];

    float yr = poly->vlist[curr_vert].x*sin_look[theta] -
              poly->vlist[curr_vert].y*cos_look[theta];

    // store result back
    poly->vlist[curr_vert].x=xr;
    poly->vlist[curr_vert].y=yr;

} // end for curr_vert

// return success
return(1);

} // end Rotate_Polygon2D

```

如果想使对象 POLYGON2D 旋转 10 度，你只需这样调用：

```
Rotate_Polygon2D(&object, 10);
```

注意



所有的旋转都将破坏原始多边形的坐标。如果你将原来的矢量正转 10 度，而后再反转 10 度，由于不断取整，会使你原来的坐标丢失，这就是为什么还需要保存一套原始坐标的原因，必要时可以取出原始坐标刷新你的数据。以后会看到许多类似的情况。

关于精度

前面我写的演示程序采用整数存储矢量，但是令人沮丧的是旋转几次就精度大降，因此现在不得不用浮点数来写演示程序。你需要将 POLYGON2D 结构改写成具有浮点精度矢量，而不是整数精度矢量。

有两种办法。一是本地坐标和旋转坐标(世界坐标)都用整数存储，将本地坐标转变成浮点数，在变换坐标中存储变换坐标，渲染。随后对于下一帧，再次使用本地坐标，那样，在本地坐标中就没有误差的积累了。

或者你干脆使用一套浮点数本地/变换坐标。我这里采用这种方法。因此，你有两套新的数据结构分别存储矢量和多边形。

```
// a 2D vertex
typedef struct VERTEX2DF_TYP
{
    float x, y; // the vertex
} VERTEX2DF, *VERTEX2DF_PTR

// a 2D polygon
typedef struct POLYGON2D_TYP
{
    int state; // state of polygon
    int num_verts; // number of vertices
    int x0, y0; // position of center of polygon
    int xv, yv; // initial velocity
    DWORD color; // could be index or PALETTEENTRY
    VERTEX2DF *vlist; // pointer to vertex list
} POLYGON2D, *POLYGON2D_PTR
```

我这里只是将矢量列表改写为浮点数方式，目的是不需要重新编写所有的东西。现在，虽然移动还是基于整数，但所有移动和旋转都可以正常工作了。当然，我可以一开始就这么写，但我想让你看到整个游戏编程的过程。你觉得可以进行了，但是有时却不行，你不得不回过头来重新开始。

为了同时看到旋转和移动，我编写了 DEMO8_3.CPP 并把它写进了 DEMO8_4.CPP\EXE。它使星形线以不同的速度旋转，程序也采用了查询表，看一下吧！

缩放

学过前面之后，任何事情都会简单了。缩放和移动一样简单。看图 8.24。你需要做的就是将每个坐标都乘以缩放因子。

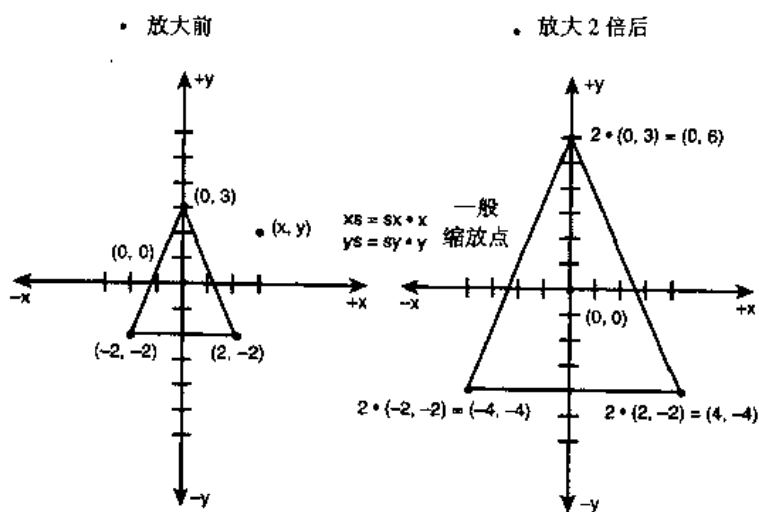


图 8.24 缩放数学

缩放因子比 1.0 大，将放大对象，比 1.0 小则将缩小对象。等于 1.0 的缩放因子不改变对象大小。数学上将点(x, y)缩放 s 倍，得到(xs, ys)，即：

```
xs=s * x
ys=s * y
```

当然，你也可以对不同轴采用不同的缩放。也就是说你可以为 x、y 坐标分别使用不同的缩放因子。就像下面这样。

```
xs=sx * x
ys=sy * y
```

但多数时候，你希望放大倍数相同。不过，谁知道呢？你也可能只让一个轴放大。这里是放大多边形的函数，对 x、y 轴分别放大。

```
int Scale_Polygon2D(POLYGON2D_PTR poly, float sx, float sy)
{
    // this function scales the local coordinates of the polygon

    // test for valid pointer
    if (!poly)
        return(0);

    // loop and scale each point
    for (int curr_vert = 0; curr_vert < poly->num_verts; curr_vert++)
    {
        // scale and store result back
        poly->vlist[curr_vert].x *=sx;
        poly->vlist[curr_vert].y *=sy;

    } // end for curr_vert

    // return success
    return(1);

} // end Scale_Polygon2D
```

哈哈，很简单！

如果想将多边形缩小到 1/10，可以这样调用：

```
Scale_Polygon2D(&polygon, 0.1, 0.1);
```

注意到 x, y 轴的放大倍数都是 0.1。因此，缩放在每个轴上均匀地进行。

作为缩放的演示程序，我给出了 DEMO8_5.CPPIEXE 程序。程序给出了一个旋转的星形线，当按下 A 时，对象放大 10%，按下 S，缩小 10%。

注 意

你可能注意到在多数演示程序中，可以看到鼠标。如果你想让它消失(对游戏来说一般是个好主意)，可以通过调用 WIN32 的 ShowCursor(BOOL bshow)函数。如果发送一个 TRUE，中间的显示计数器增加 1，发送 FALSE 则减 1。当系统开始时，显示计数器为 0，当它大于等于 0 时，就显示鼠标。因此，调用 ShowCursor(FALSE)将会去掉鼠标。之后如果再次调用 ShowCursor(TRUE)将会再次显示鼠标。但是，要记住，ShowCursor()函数积累你调用的次数，也就是说，如果你调用了 5 次 ShowCursor(FALSE)函数，你就需要再调用 5 次 ShowCursor(TRUE)才能够将之解除。

矩阵引论

当我们谈到三维图形时，我将真正把你带到矢量、矩阵和其他数学概念之中。但是，现在我先给你一些矩阵的信息，并让你看看它是怎样像你以前所做的那样被用到二维图形的变换的。

矩阵不过是给定了行和列的矩形数字阵列。我们通常提到的矩阵为 $m \times n$ 矩阵，也就是它有 m 行， n 列。 $m \times n$ 也指矩阵的维数，例如，这里有 2×2 矩阵 A:

$$A = \begin{vmatrix} 1 & 4 \\ 9 & -1 \end{vmatrix}$$

注意我用了大写字母 A 来表示矩阵，一般情况下，多数人用大写字母表示矩阵，黑体字表示矢量，在前面的例子中，第一行是 $\langle 1, 4 \rangle$ 第二行是 $\langle 9, -1 \rangle$ 。这里再给出一个 3×2 矩阵。

$$B = \begin{vmatrix} 5 & 6 \\ 2 & 3 \\ 100 & -7 \end{vmatrix}$$

这里是一个 2×3 矩阵。

$$C = \begin{vmatrix} 3 & 5 & 0 \\ -8 & 12 & 4 \end{vmatrix}$$

为了确定矩阵中的元素 $\langle i, j \rangle$ ，你只需找到 i 行 j 列的值。但是，要注意，多数程序把矩阵开始的元素记为 1，而不是在计算机程序中的 0，记住这一点，将来你从 0 开始计数，这样会使你的 C++ 矩阵更加自然。例如，这里给出了有下标的 3×3 矩阵:

$$A = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

太简单了，这里给出了实际的矩阵带有的约定成俗的下标。但是可能你会问，矩阵是从哪里来的？矩阵是懒惰的数学家们的工具，当然了，不是说你。一般，如果你有下面的等式：

$$3x + 2y = 1$$

$$4x - 9y = 9$$

这时，你的许多工作就是记下变量。你知道它们是(x, y)，那么为什么还要写它们呢？为什么不只记下有用的东西呢？这样就产生了矩阵。在前面的例子中，有三套值你可以写入矩阵，你可以一起或分开使用这些值。

下面是系数矩阵：

$$3x + 2y = 1$$

$$4x - 9y = 9$$

$$A = \begin{bmatrix} 3 & 2 \\ 4 & -9 \end{bmatrix}$$

维数为 2×2

下面是变量矩阵：

$$3x + 2y = 1$$

$$4x - 9y = 9$$

$$X = \begin{bmatrix} x \\ y \end{bmatrix}$$

维数是 2×1

最后是右边的常量矩阵：

$$3x + 2y = 1$$

$$4x - 9y = 9$$

$$B = \begin{bmatrix} 1 \\ 9 \end{bmatrix}$$

维数是 2×1

根据这些矩阵，你可以把注意力集中到系数矩阵 A 上来，另外，你也可以像下面一样写矩阵等式：

$$A \times X = B$$

如果你想进行数学计算，可以得到：

$$3x + 2y = 1$$

$$4x - 9y = 9$$

至于如何进行数学运算，下面将接着谈。

特征矩阵

在所有的数学系统中，需要首先定义的是 1 和 0。矩阵也有类似的值。和 1 相类似的是特征矩阵，在该矩阵的对角线上的元素为 1，其他元素为 0。此外由于矩阵的大小可以为任意尺寸，显然有无穷个特征矩阵。但是，有一个约束，就是特征矩阵必须为方阵。也就是说必须是 $m \times m$ 矩阵，这里 $m \geq 1$ 。下面给出两个例子。

$$I_2 = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

维数 2×2

$$I_3 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

维数 3×3

具有讽刺意味的是特征矩阵除了在矩阵相乘的时候(马上就会谈到)并不总是像 1。

另外一个基本矩阵被称为零矩阵。相加和相乘都是 0，它不过是一个 $m \times n$ 的 0 元素矩阵，没有其他特殊约束。

$$Z_{3 \times 3} = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

$$Z_{1 \times 2} = \begin{vmatrix} 0 & 0 \end{vmatrix}$$

零矩阵惟一有意思的是它在矩阵相加或矩阵相乘的时候无方向性。其他方面则没有用。

矩阵加法

矩阵的加减是将两个矩阵相对应的元素分别进行加减，结果组成的矩阵就是矩阵加减后的结果。惟一需要遵循的原则是进行加减的两个矩阵的维数要相同。这里是两个例子：

$$A = \begin{vmatrix} 1 & 5 \\ -2 & 0 \end{vmatrix} \quad B = \begin{vmatrix} 13 & 7 \\ 5 & -10 \end{vmatrix}$$

$$A+B = \begin{vmatrix} 1 & 5 \\ -2 & 0 \end{vmatrix} + \begin{vmatrix} 13 & 7 \\ 5 & -10 \end{vmatrix} = \begin{vmatrix} (1+13) & (5+7) \\ (-2+5) & (0-10) \end{vmatrix} = \begin{vmatrix} 14 & 12 \\ 3 & -10 \end{vmatrix}$$

$$A+B = \begin{vmatrix} 1 & 5 \\ -2 & 0 \end{vmatrix} - \begin{vmatrix} 13 & 7 \\ 5 & -10 \end{vmatrix} = \begin{vmatrix} (1-13) & (5-7) \\ (-2-5) & (0+10) \end{vmatrix} = \begin{vmatrix} -12 & -2 \\ -7 & 10 \end{vmatrix}$$

注意到加和减是有联系的;就是说, $A+(B+C)=(A+B)+C$, 但是减法不符合分配律, 另外, $(A-B)$ 不等于 $(B-A)$ 。

矩阵乘法

有两种形式的矩阵乘法: 标量相乘和矩阵相乘。标量的乘法是指将矩阵同一个标量数相乘。只需将每个矩阵元素乘以该标量即可。矩阵可以是 $m \times n$ 的任何维数矩阵。

下面是一个 3×3 矩阵。k 为任意实数:

$$\text{令 } A = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

$$\text{则 } kA = k \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix} = \begin{vmatrix} ka_{00} & ka_{01} & ka_{02} \\ ka_{10} & ka_{11} & ka_{12} \\ ka_{20} & ka_{21} & ka_{22} \end{vmatrix}$$

下面是一个例子:

$$3 \begin{vmatrix} 1 & 4 \\ -2 & 6 \end{vmatrix} = \begin{vmatrix} (3 \times 1) & (3 \times 4) \\ (3 \times (-2)) & (3 \times 6) \end{vmatrix} = \begin{vmatrix} 3 & 12 \\ -6 & 18 \end{vmatrix}$$

数 学

标量乘法对矩阵等式也有效, 只要你在两边同时相乘即可。这是事实, 因为你在任何时候在等式的左边和右边同时乘以一个常数, 等式不变。

第二类矩阵相乘是真正的矩阵与矩阵相乘。数学运算有点复杂, 不过你可以将一个矩阵作为操作数, 对另外一个矩阵操作。假设有两个矩阵 A 和 B, 你想把它们相乘, 它们一定要有相同的内部维数。换句话说, 如果 A 是 $m \times n$ 阵, 则 B 应是 $n \times r$ 阵。m、r 可以相等, 也可以不等。但是内部维数必须相等。例如: 你可以把 2×2 矩阵乘以 2×2 矩阵, 3×2 矩阵乘以 2×3 矩阵, 4×4 矩阵乘以 4×5 矩阵, 但是不能够 3×3 矩阵乘以 2×4 阵, 因为内部维数不同。相乘结果的维数为乘数和被乘数的外部维数。例如: 2×3 阵乘以 3×4 阵, 结果是 2×4 阵。

矩阵相乘是很难用语言描述的事。我总是不得不举例子进行说明, 在我对乘法运算进行解释时, 请看图 8.25。假设有矩阵 A 和 B, 现在计算 $A \times B$ 的结果矩阵 C 的各个元素,

需要将 A 矩阵的 1 行与 B 矩阵的 1 列相乘。将个元素相乘结果结果相加，作为 C 矩阵的元素。这里给出一个 2×2 乘以 2×3 矩阵的例子。

$$\text{令 } A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 3 & 5 \\ 6 & 0 & 4 \end{bmatrix}$$

$$C = A \times B = \begin{bmatrix} (1 \times 1 + 2 \times 6) & (1 \times 3 + 2 \times 0) & (1 \times 5 + 2 \times 4) \\ (3 \times 1 + 4 \times 6) & (3 \times 3 + 4 \times 0) & (3 \times 5 + 4 \times 4) \end{bmatrix} = \begin{bmatrix} 13 & 3 & 13 \\ 27 & 9 & 31 \end{bmatrix}$$

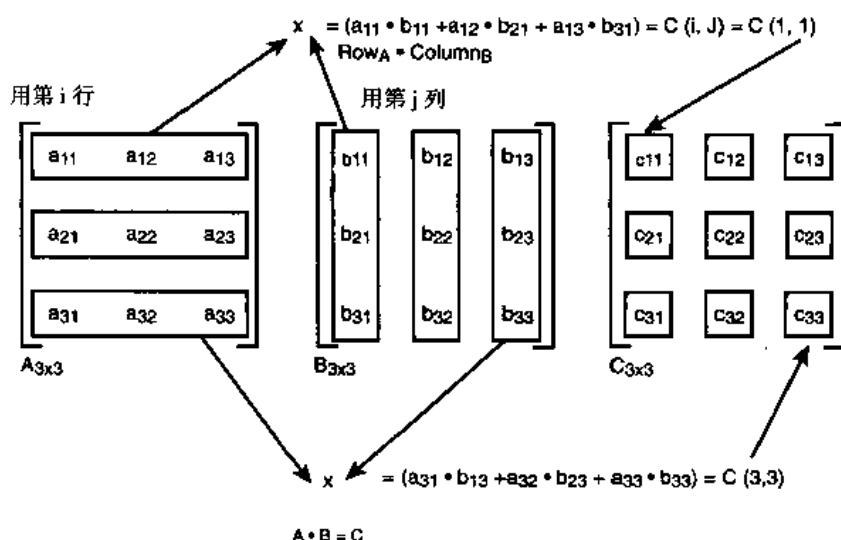


图 8.25 矩阵乘法的机理

作为例子，请注意黑体部分 $(1 \times 1 + 2 \times 6)$ 。它的结果以及其他所有的结果都是矢量的点积（矢量像矩阵一样也是数值，不过是一行）。点积的数学意义很明确，就是将两个 $1 \times n$ 矩阵的各个对应元素的乘积求和的结果。或者在数学上如下所示：

$$\text{令 } a = [1 \ 2 \ 3] \quad b = [4 \ 5 \ 6]$$

$$a \cdot b = [(1 \times 4) + (2 \times 5) + (3 \times 6)] = 32$$

或者，如果想更准确点，只是一个缩放而已。

警告



我现在对点积有点彬彬有礼，技术上来说，它们只是对矢量有效，但是实际上矩阵的列或行实际上就是矢量。基本上，我是在帮你而不是把你领入歧途。

这就是你如何计算矩阵乘法，另外一个计算 $A \times B$ 阵的结果 C 阵的方法是一个一个元素求解。就是说，如果你想得出 c_{ij} 元素（这里 i, j 从 0 开始），你可以通过 A 矩阵的 i 行和 B 矩阵的 j 列的点积（将结果相加）来求得。

至此，我想你对矩阵的乘法有了基本的了解。让我们来看一些矩阵相乘的实例吧。首先，我们来定义矩阵类型。

```
// here's a 3x3, useful for 2D stuff and some 3D
typedef struct MATRIX3X3_TYP
{
    float M[3][3]; // data storage
} MATRIX3X3, *MATRIX3X3_PTR;

int Mat_Mul3X3(MATRIX3X3_PTR ma,
               MATRIX3X3_PTR mb,
               MATRIX3X3_PTR mprod)
{
    // this function multiplies two matrices together and
    // stores the result

    for (int row=0; row<3; row++)
    {
        for (int col=0; col<3; col++)
        {
            // compute dot product from row of ma
            // and column of mb

            float sum = 0; // used to hold result

            for (int index=0; index<3; index++)
            {
                // add in next product pair
                sum += (ma->M[row][index]*mb->M[index][col]);
            } // end for index

            // insert resulting row,col element
            mprod->M[row][col] = sum;

        } // end for col

    } // end for row

    return(1);

} // end Mat_Mul3X3
```

你会注意到有许多数学运算。一般的，矩阵乘法是三次方运算，也就是说需要三重循环，当然也可以进行优化，如测试乘数和被乘数是不是零，如果是零就可以直接跳过去，不进行乘法运算。

利用矩阵进行变换

采用矩阵进行二维或三维变换很简单，基本上，你只需将想进行变换的点同变换矩阵相乘即可，或者说：

$$p' = p \times M$$

假设 p' 是变换后的点， p 是原来的点， M 是变换矩阵，如果我还没有给你讲过矩阵乘法没有交换律，现在再给你讲一遍。

$(A \times B)$ 不等于 $(B \times A)$

当然这种说法一般都是正确的，除非 A 阵或 B 阵是特征矩阵，或者是零矩阵，或者它们是同一个矩阵。当进行矩阵运算时注意顺序。

下面你将把一个 (x, y) 点变成一个 1×3 的行矩阵，之后将它同 3×3 变换矩阵相乘。结果是一个 1×3 行矩阵，但是，你虽然可以得到前两个元素 x' , y' ，如何得到第三个必要的元素呢？

通常你可以这样来处理所有的点：

$[x, y, 1.0]$

元素 1.0 称为归一化坐标，这就使任何变换点可以缩放，并允许进行移动变换。除了这一点，其数学意义并不重要。可以把它看成你所需要的哑元变量。所以你创建了一个 1×3 矩阵，之后就可以同你的变换矩阵相乘了，这里是点或称为 1×3 矩阵的数据结构：

```
typedef struct MATRIX1X3_TYP
{
    float M[3]; // data storage
} MATRIX1X3, *MATRIX1X3_PTR
```

下面是一个点乘以一个 3×3 的矩阵的函数：

```
int Mat_Mul1X3_3X3(MATRIX1X3_PTR ma,
                   MATRIX3X3_PTR mb,
                   MATRIX1X3_PTR mprod)
{
    // this function multiplies a 1X3 matrix against a
    // 3X3 matrix-ma*mb and store the result

    for (int col = 0; col<3; col++)
    {
        // compute dot product from row of ma
        // and column of mb

        float sum = 0; // used to hold result

        for (int index=0; index<3; index++)
        {
```

```

        // add in next product pair
        sum += (ma->M[index]*mb,>M[index][col])
    } // end for index

    // insert resulting col element
    mprod->M[col] =sum;

    } // end for col
return(1);

} // end Mat_Mul_1X3_3X3

```

建立一个坐标为 x, y 的点 p , 你需要这样做:

```
MATRIX1X3 = {x, y, 1};
```

记住所有这些, 让我们来看一个你已经手工进行过的变换矩阵。

变换

进行变换, 你想在进行矩阵相加时只进行 x, y 的运算, 矩阵会自动做到这点。

令 $Mt = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 0 \end{vmatrix}$

例如:

$p = [x \ y \ 1]$

$p' = p \times Mt = [x \ y \ 1] \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 0 \end{vmatrix} = [(x+1 \times dx) \ (y+1 \times dy) \ 1]$

数 学

注意左边矩阵中需要因子 1.0。没有它, 就根本不能够进行转换。

并且, 如果你抽出前两个元素, 你就会得到:

$x' = x + dx$

$y' = y + dy$

这正是你所需要的。

缩放

相对于原点进行点的缩放，需要将坐标 x 、 y 分别乘以缩放因子 s_x 、 s_y 。另外在你进行缩放时你不想有移动存在，这里给出了你想要的矩阵：

$$M_s = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Eg. } p = [x \ y \ 1]$$

$$p' = p \times M_s = [x \ y \ 1] \times \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = [(x \times s_x) \ (y \times s_y) \ 1]$$

再次说明，缩放后的结果是：

$$x' = s_x \times x$$

$$y' = s_y \times y$$

数 学

注意转换矩阵右下方的 1。技术上来说不需要它，因为它对缩放不起作用。因为你根本就不需要第三列。问题是能够将第三列去掉吗？在回答问题之前，让我们来看看旋转吧。

旋转

旋转矩阵是所有的变换矩阵中最复杂的，因为它的里面都是函数，基本上我们想通过旋转公式来旋转输入点，为了做到这点，你必须观察旋转公式，提取出操作符，把它们压缩进一个矩阵中。另外，你也不想有任何移动，所以最后一行的前两个元素的值为零。矩阵是按如下方式来工作的。

$$M_r = \begin{bmatrix} \cos x & -\sin x & 0 \\ \sin x & \cos x & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Eg. } p = [x \ y \ 1]$$

$$p' = p \times M_r = [x \ y \ 1] \times \begin{bmatrix} \cos x & -\sin x & 0 \\ \sin x & \cos x & 0 \\ 0 & 0 & 1 \end{bmatrix} = [(x \times \cos x - y \times \sin x) \ (x \times \sin x + y \times \cos x) \ 1]$$

非常正确。

在讨论多边形的移动之前，让我们来讨论一下采用 3×2 矩阵而不是采用 3×3 矩阵的问题，看起来在所有的矩阵乘法运算中最后一项被忽略了，另外总是 1.0。这两种说法都正确。

因此，你做的变换如果仅限于此，你可以采用 3×2 矩阵，但是，我更乐意采用 3×3 矩阵使点和坐标均一化。1.0(实际上可以看成是 q)的重要性在于：为了使坐标在变换完成后保证有正确的格式，你必须除以参数 q ，也即：

$$p'=[x \ y \ q]$$

$$x'=x/q$$

$$y'=y/q$$

但是由于这时 $q=1$ ，不需要进行除法运算，另外，在讨论三维图形变换时参数还有重要的用途，记住这点。

在任何时候，根据上面的信息，你可以改变多个数据的结构，把所有的点存储为 1×2 矩阵，变换矩阵采用 3×2 矩阵，可以采用下面的数据结构和变换函数。

```
// the transformation matrix

typedef struct MATRIX3X2_TYP
{
    float M[3][2]; // data storage
} MATRIX3X2, *MATRIX3X2_PTR;

// our 2D point
typedef struct MATRIX1X2_TYP
{
    float M[2]; // data storage
} MATRIX1X2, *MATRIX1X2_PTR;

int Mat_Mul1X2_3X2(MATRIX1X2_PTR ma,
                   MATRIX3X2_PTR mb,
                   MATRIX1X2_PTR mprod)
{
    // this function multiplies a 1x2 matrix against a
    // 3x2 matrix - ma*mb and stores the result
    // using a dummy element for the 3rd element of the 1x2
    // to make the matrix multiply valid i.e. 1x3 X 3x2

    for (int col=0; col<2; col++)
    {
        // compute dot product from row of ma
        // and column of mb

        float sum = 0; // used to hold result
```

```

for (int index=0; index<2; index++)
{
    // add in next product pair
    sum+=(ma->M[index]*mb->M[index][col]);
} // end for index

// add in last element * 1
sum+= mb[index][col];

// insert resulting col element
mprod->M[col] = sum;

} // end for col

return(1);

} // end Mat_Mul_1X2_3X2

```

注 意

你必须将一个 $m \times r$ 矩阵同一个 $r \times n$ 矩阵相乘，也就是说，内部维数必须相同。很明显， 1×2 矩阵不能和 3×2 矩阵相乘，因为 2 不等于 3。但是在代码中，你可以加一个 1.0 哑元到 1×2 矩阵中，使它变成 1×3 矩阵，这样做仅仅是为了数学运算。

DEMO8_6.CPP\EXE 给出了矩阵用法的演示程序。我用线框画出了一个类似飞机的多边形，你可以对它进行缩放、旋转、移动。如图 8.26 所示。

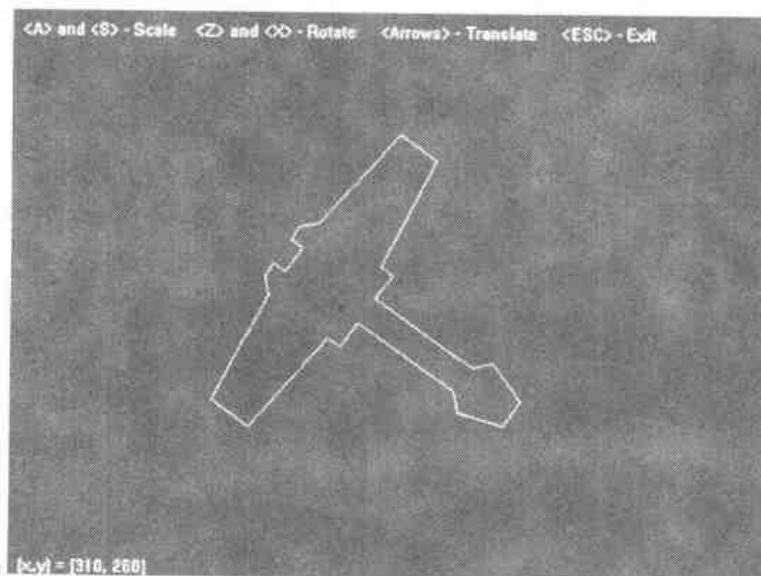


图 8.26 运行 DEMO8_6.EXE

这里是演示程序的控制键。

Esc	退出演示程序
A	放大 10%

S	缩小 10%
Z	逆时针旋转
X	顺时针旋转
箭头键	移动

填充实心多边形

让我们结束数学讨论来回到实际的东西上来吧。最实用的三维引擎和许多二维引擎，是画实体或填充多边形，如图 8.27 所示。这是你需要解决的下一个问题。

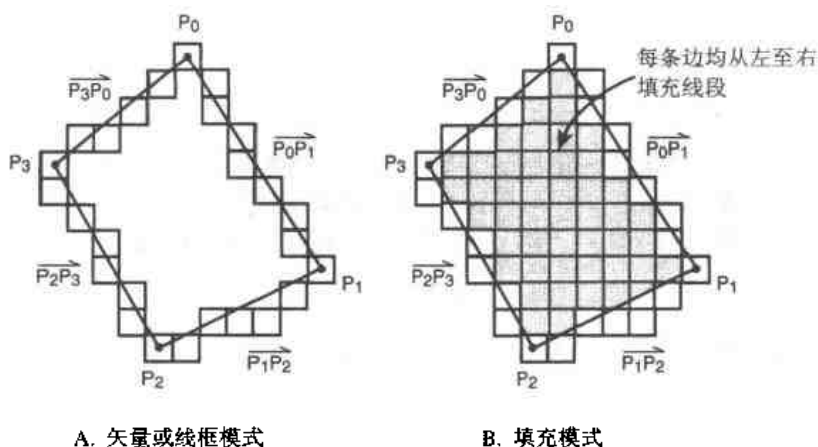


图 8.27 填充多边形

有很多方法画填充多边形，但是，因为所有这些点都是为了创建二维或三维游戏，所以你想画的填充多边形有单一颜色或贴字，如图 8.28 所示。现在让我们不考虑贴字（等到三维的内容时再考虑），只考虑如何画任意颜色的实多边形。

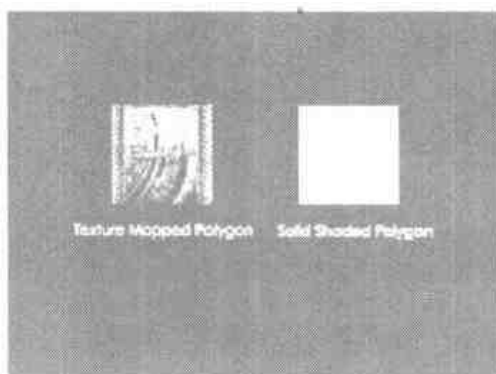


图 8.28 实多边形和文字映射多边形对比

在解决问题之前，你必须明确你要解决什么问题。首先的约束是多边形必须为凸多边形

形。中间没有洞或者形状不奇异。之后你不得不确定多边形到底多复杂，它可以由三条、四条、任意数量的直线组成么？这是一个明确的问题，你必须对边数多于三条的多边形采用不同的算法（四边形可以分割成两个三角形）。

因此，我将给出如何进行普通多边形和三角形的填充（这将是创建三维引擎的基础）。

三角形和四边形

首先，让我们来看看普通的四边形，如图 8.29 所示。四边形可以被分解为两个三角形， ta 和 tb ，这样可以简化绘制四边形的问题。因此，现在你可以把注意力集中在画一个三角形上来，既可以用它画三角形又可以用来画四边形。如图 8.30 所示，让我们开始吧。

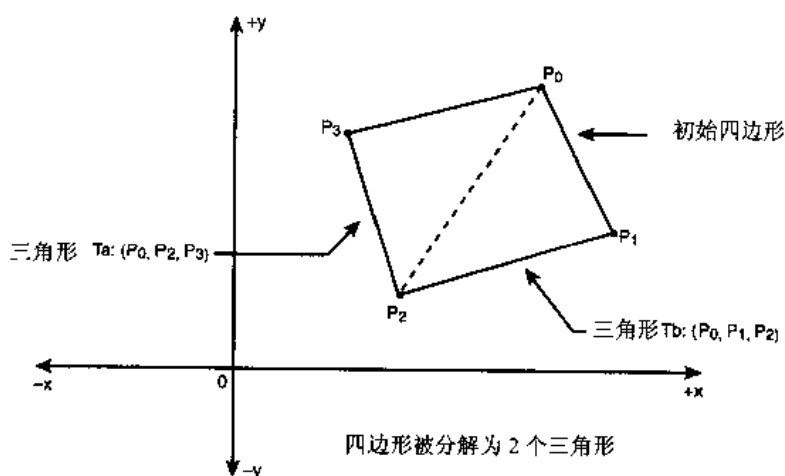


图 8.29 通用的四条边的四边形

任意三角形都属于 4 种类型之一

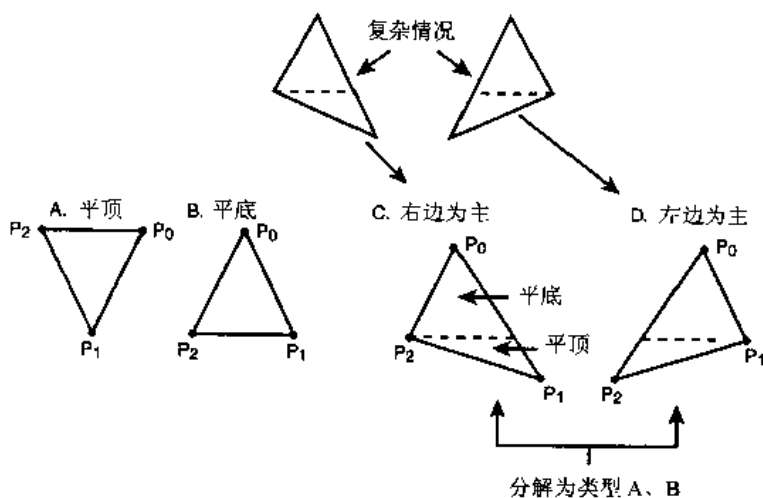


图 8.30 通用的三角形

首先, 你可能创建的三角形只有四种类型。将它们表示为:

平顶: 这是具有平顶的三角形, 换句话说就是最上面的两个顶点具有相同的 y 坐标。

平底: 这是具有平底的三角形, 换句话说就是最下面的两个顶点具有相同的 y 坐标。

右边为主: 这样的三角形的三个顶点的 y 坐标各不相同, 但是最长边在右侧。

左边为主: 这样的三角形的三个顶点的 y 坐标各不相同, 但是最长边在左侧。

前两种最容易进行光栅化, 因为两条三角形的腿一样长(你马上就可以看到这点的重要性)。但是如果你把后两种分割成平顶或平底的一对三角形, 它们也很简单。不知你是否想这样做, 但是我常常这样做。如果不这样做, 你的光栅化过程就不得不包含许多判断逻辑用于光栅化。总之, 如果你看一些例子就会更加明白。

绘制三角形和四边形

画三角形和画线十分类似, 因为你必须辨别三角形的两个边界并用直线一条一条地填充。图 8.31 给出了平底三角形的填充示意图。如你所见, 一旦每条边的斜率被计算出来, 你就可以把扫描线向下移, 根据斜率调整端点坐标 x 的变化量 x_s 、 x_e (更精确一点是 $1/\text{斜率}$), 然后画连接线。

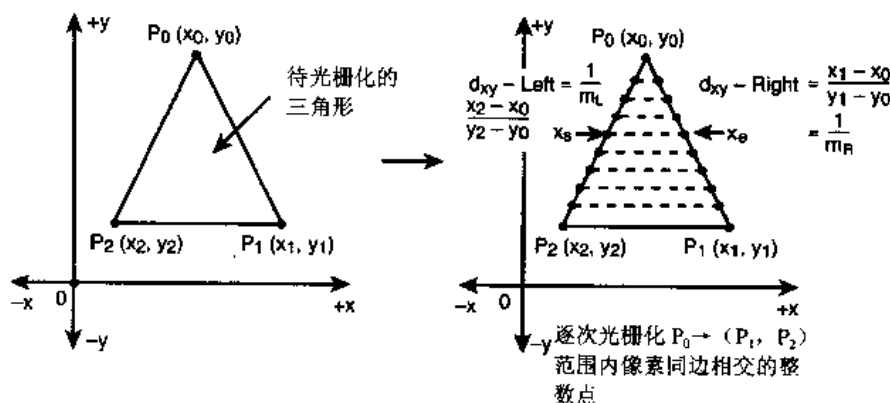


图 8.31 光栅化三角形

你不必使用 Bresenham 算法, 因为你对画线没有兴趣。你所关心的是直线同每个整数步长的像素中心的相交。下面给出平底三角形填充算法。

1. 首先计算左边和右边的斜率。基本上是 $1/\text{斜率}$ 。你需要它是因为你要用垂直定位法。因此, 你想知道每单位 y 引起 x 的变化量。简单说就是 dx/dy 或 $1/M$ 。对于左边或右边, 分别称之为: dxy_left 和 dxy_right 。
2. 从最高点 (x_0, y_0) 开始, 使 $x_s = x_e = x_0$ 且 $y = y_0$ 。
3. 将 x_s 加上 dxy_left , x_e 加上 dxy_right 。这样跟踪要填充的每个端点。
4. 从 (x_s, y) 向 (x_e, y) 画线。
5. 回到第三步, 直到从顶端到底部都被光栅化。

当然，算法的起始条件和边界情况都需要特别注意，但是一旦做完，或多或少，其实都相当简单，现在，让我们在做其他事情之前先看看优化问题吧。

看一眼，你可能就想对边界使用浮点数，它会工作得更好。但是，问题并不在于奔腾处理器上浮点数要比整数运算慢，而是有时不得不将浮点数变换为整数。

如果你让编译器做这件事，需要大概 60 个周期。如果用 FPU 码手工计算，需要 10~20 个周期，（记住，你需要变换成整数并进行存储）。因此，我拒绝仅仅在光栅化每条直线时找端点就损失 60 个周期的做法。因此，我只是用浮点数的版本给你演示算法，而最终的模型将采用浮点-整数混合型（稍后将给你介绍）。

让我们来完成平底三角形的基于浮点型数据的光栅化过程。首先，我们如图 8.31 那样标注，这里给出算法：

```
// compute deltas
float dxy_left  = (x2-x0)/(y2-y0);
float dxy_right = (x1-x0)/(y1-y0);

// set starting and ending points for edge trace
float xs =x0;
float xe =x0;

// draw each scanline
for (int y=y0; y<=y1; y++)
{
    // draw a line from xs to xe at y in color c
    Draw_Line((int)xs, (int) xe, y, c);

    // move down one scanline
    xs+=dxy_left;
    xe+=dxy_right;

} // end for y
```

现在，让我们来谈谈算法的详细情形，并看看什么被忘记了。首先，算法脱离了每个扫描线的端点，这是比较糟糕的事情，因为你在丢失有用的信息，一个较好的解决方法是将每个端点变换成整数之前加上 0.5。另外一个问题和初始条件有关，在开始时，算法画了一个单像素宽的直线，这有一定作用，但明显还有进行优化的空间。

现在，让我们看看你能不能够基于你所了解的东西编写平顶三角形算法程序，你所需要做的是如图 8.31 所示那样标出坐标，然后改变边界条件，使左边和右边的内插值正确计算。下面是所做的改动：

```
// compute deltas
float dxy_left = (x2-x0)/(y2-y0);
float dxy_right = (x1-x0)/(y1-y0);
```

```
// set starting and ending points for edge trace
float xs = -x0;
float xe = -x1;

// draw each scanline
for (int y=y0; y<=y2; y++)
{
    // draw a line from xs to xe at y in color c
    Draw_Line((int)(xs+0.5), (int)(xe+0.5), y, c);

    // move down one scanline
    xs+=dxy_left;
    xe+=dxy_right;
} // end for y
```

哪个好些？现在回到实际中来，你毕竟才完成了一半。目前，你知道如何画一个平顶的三角形，并知道不是平顶的三角形可以被分解为平顶的三角形。让我们来看看如何处理。

图 8.32 给出了一个右边为主的三角形，不用证明，如果你能够进行右边为主的三角形的光栅化，左边为主的三角形的光栅化就微不足道了。首先应该注意的是，你的算法需要从平底三角形的绘制开始，也就是说，从同一个起始点进行边的内插。问题会在左边内插到第二个顶点时出现，这是你需要进行改动的地方。从根本上说，你需要重新计算左边的内插值，然后再进行光栅化。

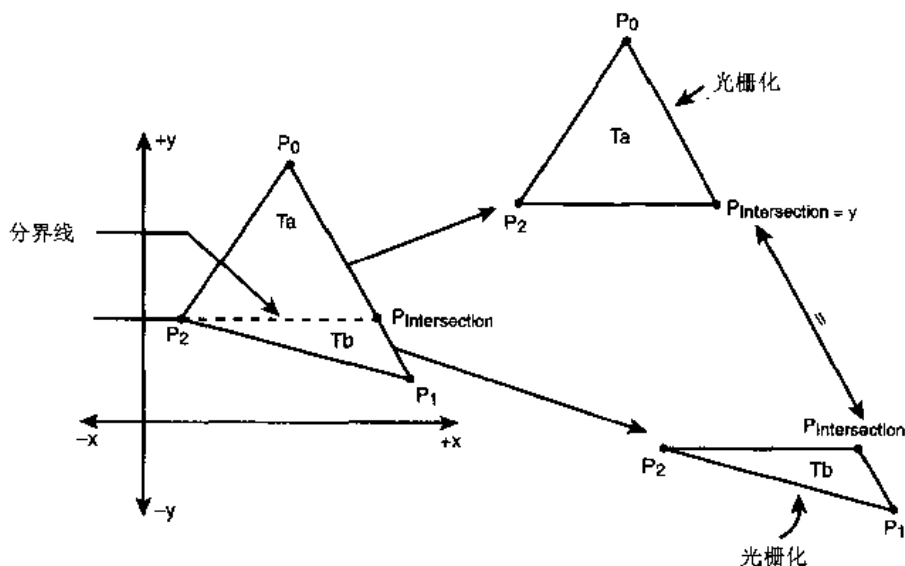


图 8.32 右边为主三角形的光栅化

有许多解决这个问题的办法，在内循环时，你可以首先画三角形的第一部分，直到斜率改变，重新计算内插值再继续。或者你可以将三角形分成一个平顶三角形和一个平底三角形，然后调用你已经知道的画平顶三角形和平底三角形的代码。

目前先使用第二种办法,如果将来你发现在三维技术中这种方法不能够满足,可以采用上一种办法或其他方法。

三角形分解详述

在我给出画一个8位着色的三角形程序代码之前,我想首先和你谈谈编写算法程序时的一些细节。

把三角形进行分解,分成一个平顶三角形和一个平底三角形,有一定的技巧。你需要确定短边的高度(长度),直到斜率发生改变,通过短边找出长边上分割三角形的点。基本上,你可以利用三角形顶点的垂直跨度,随后不是采用对长边内插一条直线,而是一次用乘法内插 n 条扫描线。

结果就如同人工沿着三角形的长边一个扫描线一个扫描线的移动。然而,一旦找到了分边的正确点,你就可以调用你的平顶或平底三角形的光栅化程序画出这个三角形。图8.32给出了分割算法的具体详细情况。

除了分割三角形,还有其他的一点小问题,就是重画现象。如果平顶三角形和平底三角形有一个公共顶点,两者公用的那条线就会被光栅化两次。这不是什么大问题,但是需要考虑到。你可以在平底三角形底部跳过一条扫描线,避免公用扫描线的重画。

差不多了,让我们来看看,还有什么。是的,什么是分割?如果你回忆一下,可知有两种方法进行分割:对象分割和图像分割。对象分割很伟大,但是,如果将三角形同屏幕矩形相割,最坏的情况下将会增加额外的4个顶点,如图8.33所示。

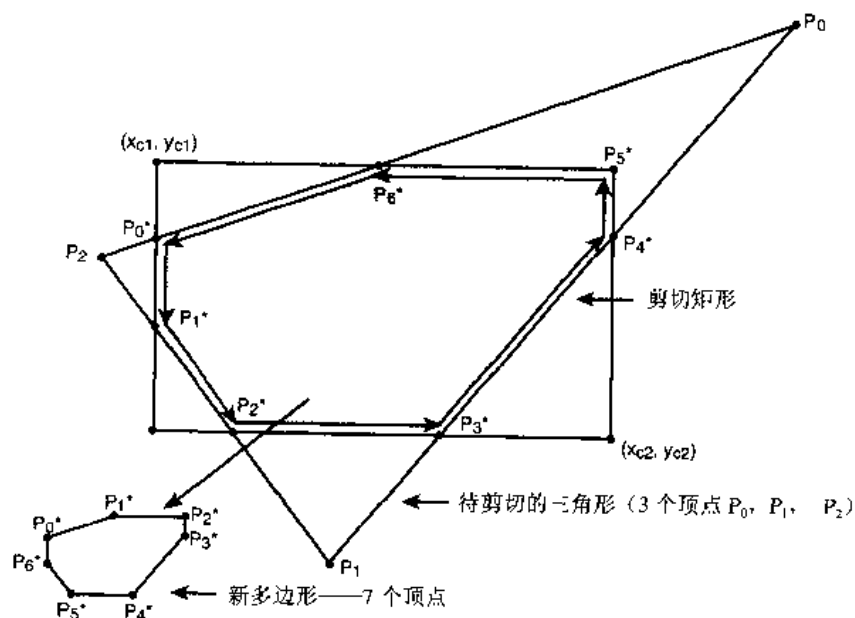


图 8.33 剪切三角形时最坏的情况

你不必采用一种新的简单办法，如画多边形那样用图形分割进行剪切，但至少应该剪切每条扫描线，而不是剪切像素点。另外，还需要进行一些简单的拒绝判断，以决定剪切是否必要。如果没有必要，你可以跳到没有剪切判断的代码处运行，以便运行更快一些，听起来不错吧？

最后，让我们讨论一下琐碎的拒绝和检测，我们需要定下来分割三角形的每种情形。如单个点、水平线或垂直线，代码在判断这种情况时不能失效。当然，我们也不能够假设我们给函数提供的顶点具有正确的顺序，所以你需要对它们从上至下，从下至右排序，那样你就可以得到一个已知的起始点，记住这点，这里给出组成你的 8 位三角形引擎的三个函数：

这个是画平顶三角形的函数：

```
void Draw_Top_Tri(int x1,int y1,
                  int x2,int y2,
                  int x3,int y3,
                  int color,
                  UCHAR *dest_buffer, int mempitch)
{
    // this function draws a triangle that has a flat top

    float dx_right,    // the dx/dy ratio of the right edge of line
          dx_left,     // the dx/dy ratio of the left edge of line
          xs,xe,       // the starting and ending points of the edges
          height;      // the height of the triangle

    int temp_x,        // used during sorting as temps
        temp_y,
        right,         // used by clipping
        left;

    // destination address of next scanline
    UCHAR *dest_addr = NULL;

    // test order of x1 and x2
    if (x2 < x1)
    {
        temp_x = x2;
        x2     = x1;
        x1     = temp_x;
    } // end if swap

    // compute delta's
    height = y3-y1;

    dx_left  = (x3-x1)/height;
    dx_right = (x3-x2)/height;

    // set starting points
```

```

xs = (float)x1;
xe = (float)x2+(float)0.5;

// perform y clipping
if (y1 < min_clip_y)
{
    // compute new xs and ys
    xs = xs+dx_left*(float)(-y1+min_clip_y);
    xe = xe+dx_right*(float)(-y1+min_clip_y);

    // reset y1
    y1=min_clip_y;

    } // end if top is off screen

if (y3>max_clip_y)
    y3=max_clip_y;

// compute starting address in video memory
dest_addr = dest_buffer+y1*mempitch;

// test if x clipping is needed
if (x1>=min_clip_x && x1<=max_clip_x &&
    x2>=min_clip_x && x2<=max_clip_x &&
    x3>=min_clip_x && x3<=max_clip_x)
{
    // draw the triangle
    for (temp_y=y1; temp_y<=y3; temp_y++,dest_addr+=mempitch)
    {
        memset((UCHAR *)dest_addr+(unsigned int)xs,
            color,(unsigned int)(xe-xs+1));

        // adjust starting point and ending point
        xs+=dx_left;
        xe+=dx_right;

    } // end for

} // end if no x clipping needed
else
{
    // clip x axis with slower version

    // draw the triangle
    for (temp_y=y1; temp_y<=y3; temp_y++,dest_addr+=mempitch)
    {
        // do x clip
        left = (int)xs;
        right = (int)xe;

        // adjust starting point and ending point
    }
}

```

```

        xs+=dx_left;
        xe+=dx_right;
<B$!~graphics;triangles;drawing>
        // clip line
        if (left < min_clip_x)
        {
            left = min_clip_x;

            if (right < min_clip_x)
                continue;
        }

        if (right > max_clip_x)
        {
            right = max_clip_x;

            if (left > max_clip_x)
                continue;
        }

        memset((UCHAR *)dest_addr+(unsigned int)left,
                color,(unsigned int)(right-left+1));

    } // end for

} // end else x clipping needed
} // end Draw_Top_Tri

```

下面是画平底三角形的函数：

```

void Draw_Bottom_Tri(int x1,int y1,
                    int x2,int y2,
                    int x3,int y3,
                    int color,
                    UCHAR *dest_buffer, int mempitch)
{
    // this function draws a triangle that has a flat bottom

    float dx_right,    // the dx/dy ratio of the right edge of line
          dx_left,     // the dx/dy ratio of the left edge of line
          xs,xc,        // the starting and ending points of the edges
          height;       // the height of the triangle

    int temp_x,         // used during sorting as temps
        temp_y,
        right,          // used by clipping
        left;

    // destination address of next scanline

```



```

    UCHAR *dest_addr;

    /* test order of x1 and x2
    if (x3 < x2)
    {
        temp_x = x2;
        x2      = x3;
        x3      = temp_x;
    } // end if swap

    // compute delta's
    height = y3-y1;

    dx_left = (x2-x1)/height;
    dx_right = (x3-x1)/height;

    // set starting points
    xs = (float)x1;
    xe = (float)x1; // +(float)0.5;

    // perform y clipping
    if (y1<min_clip_y)
    {
        // compute new xs and ys
        xs = xs+dx_left*(float)(-y1+min_clip_y);
        xe = xe+dx_right*(float)(-y1+min_clip_y);

        // reset y1
        y1 = min_clip_y;

    } // end if top is off screen

    if (y3>max_clip_y)
        y3=max_clip_y;

    // compute starting address in video memory
    dest_addr = dest_buffer+y1*mempitch;

    // test if x clipping is needed
    if (x1> min_clip_x && x1<=max_clip_x &&
        x2>=min_clip_x && x2<=max_clip_x &&
        x3>=min_clip_x && x3<=max_clip_x)
    {
        // draw the triangle
        for (temp_y=y1; temp_y<=y3; temp_y++,dest_addr+=mempitch)
        {
            memset((UCHAR *)dest_addr,(unsigned int)xs,
                color,(unsigned int)(xe-xs+1));

            // adjust starting point and ending point
            xs+=-dx_left;

```

```

        xe+=dx_right;

    } // end for

} // end if no x clipping needed
else
{
    // clip x axis with slower version

    // draw the triangle

    for (temp_y=y1; temp_y<=y3; temp_y++,dest_addr+=mempitch)
    {
        // do x clip
        left = (int)xs;
        right = (int)xe;

        // adjust starting point and ending point
        xs+=dx_left;
        xe+=dx_right;

        // clip line
        if (left < min_clip_x)
        {
            left = min_clip_x;

            if (right < min_clip_x)
                continue;
        }

        if (right > max_clip_x)
        {
            right = max_clip_x;

            if (left > max_clip_x)
                continue;
        }

        memset((UCHAR *)dest_addr+(unsigned int)left,
            color,(unsigned int)(right-left+1));

    } // end for

} // end else x clipping needed

} // end Draw_Bottom_Tri

```

下面是画任意三角形的函数，必要时将三角形剪切成一个平顶三角形和一个平底三角形：

```

void Draw_Triangle_2D(int x1,int y1,
                      int x2,int y2,
                      int x3,int y3,
                      int color,
                      UCHAR *dest_buffer, int mempitch)
{
// this function draws a triangle on the destination buffer
// it decomposes all triangles into a pair of flat top, flat bottom

int temp_x, // used for sorting
    temp_y,
    new_x;

// test for h lines and v lines
if ((x1==x2 && x2==x3) || (y1==y2 && y2==y3))
    return;

// sort p1,p2,p3 in ascending y order
if (y2<y1)
{
    temp_x = x2;
    temp_y = y2;
    x2     = x1;
    y2     = y1;
    x1     = temp_x;
    y1     = temp_y;
} // end if

// now we know that p1 and p2 are in order
if (y3<y1)
{
    temp_x = x3;
    temp_y = y3;
    x3     = x1;
    y3     = y1;
    x1     = temp_x;
    y1     = temp_y;
} // end if

// finally test y3 against y2
if (y3<y2)
{
    temp_x = x3;
    temp_y = y3;
    x3     = x2;
    y3     = y2;
    x2     = temp_x;
    y2     = temp_y;
} // end if

```

```
// do trivial rejection tests for clipping
if ( y3<min_clip_y || y1>max_clip_y ||
    (x1<min_clip_x && x2<min_clip_x && x3<min_clip_x) ||
    (x1>max_clip_x && x2>max_clip_x && x3>max_clip_x) )
    return;

// test if top of triangle is flat
if (y1==y2)
{
    Draw_Top_Tri(x1,y1,x2,y2,x3,y3,color, dest_buffer, mempitch);
} // end if
else
if (y2==y3)
{
    Draw_Bottom_Tri(x1,y1,x2,y2,x3,y3,color, dest_buffer, mempitch);
} // end if bottom is flat
else
{
    // general triangle that's needs to be broken up along long edge
    new_x = x1 + (int)(0.5+(float)(y2-y1)*(float)(x3-x1)/(float)(y3-y1));

    // draw each sub-triangle
    Draw_Bottom_Tri(x1,y1,new_x,y2,x2,y2,color, dest_buffer, mempitch);
    Draw_Top_Tri(x2,y2,new_x,y2,x3,y3,color, dest_buffer, mempitch);

} // end else

} // end Draw_Triangle_2D
```

使用函数时，你只需要调用最后一个，因为它内部调用其他的支持函数。这里是一个利用此函数用颜色 30 画坐标为(100, 100), (200, 150), (40, 200)的三角形的例子。

```
Draw_Triangle_2D(100, 100, 200, 150, 40, 200, 30, back_buffer, back_pitch);
```

一般情况下，你应该按逆时针顺序输入坐标。否则，现在虽然没有问题，但到了三维，这些细节就变得非常重要，因为在三维算法中，要根据顶点的顺序来确定多边形的前后面。

技巧



除了画多边形的函数外，我还创建了具有一些画具有一定顶点数目图形的函数，它们在光栅化时运行更快，它们也在库函数 T3DLIB1.CPP 文件中。它们虽然被分别命名为 FP+函数名，但它们的用法相同。一般你只需要调用函数 Draw_triangleFP_2D()。函数产生的效果和函数 Draw_Triangle_2D()相同，但是它运行得更快。如果你对定点数学感兴趣，跳到第十一章“算法、数据结构、内存管理及多线程”，那里进行了优化设计。

程序 DEMO8_7.CPPIEXE 给出了函数画多边形的一个例子。它以 8 位模式画了一些随机剪切的三角形。注意，全局剪切域是由普通的矩形剪切变量定义的。

```
int min_clip_x = 0, // clipping rectangle
    max_clip_x = (SCREEN_WIDTH-1),
    min_clip_y = 0,
    max_clip_y = (SCREEN_HEIGHT-1);
```

现在，让我们回到多于三个顶点的复杂多边形的光栅化技术这个主题上。

光栅化一个四边形的一般情况

如你所见，光栅化画一个三角形不是最容易的，因而，光栅化一个多于三个定点的多边形就更难了。假如你把一个四边形分成两个三角形，就不是太难，例如图 8.29 就是把一个四边形分成了两个三角形。基本上，你可以通过简单的判断算法，很容易的将四边形分解成两个三角形。

假如多边形的顶点 0, 1, 2, 3 以一定的顺序排列，如 CW（顺时针）。

三角形 1 由顶点 0, 1, 2 组成。

三角形 2 由顶点 1, 2, 3 组成。

记住这些，你就可以利用前面的代码，加入到分割程序中，进行四边形的光栅化。我已经在程序 Draw_QuadFP_2D()中替你完成了这些。它还不是一个浮点数版本，在这里给出它们的代码。

```
inline void Draw_QuadFP_2D(int x0, int y0,
                           int x1, int y1,
                           int x2, int y2,
                           int x3, int y3
                           int color,
                           UCHAR *dest_buffer, int mempitch)
{
    // this function draws a 2D quadrilateral

    // simply call the triangle function 2x, let it do all the work
    Draw_TriangleFP_2D(x0,y0,x1,y1,x2,y2,color,dest_buffer,mempitch);
    Draw_TriangleFP_2D(x1,y1,x2,y2,x3,y3,color,dest_buffer,mempitch);

} // end Draw_QuadFP_2D
```

函数同三角形函数完全一样，不过是多了一个顶点，函数演示程序为 DEMO8_8.CPP\EXE。它在屏幕上随机创建一些四边形。

注意



我在这里对参数处理的较为简单，你可以通过定义一个多边形结构为函数传递一个地址，而不是一系列数据。现在我先讲到这里，但是要记住这点，因为当到了 3D 时，你还会遇到它。

四边形的三角形分割

那么，你可以通过三角形的光栅化来光栅化四边形，但是如何进行多于四边的多边形的图形的光栅化呢？你可以如图 8.34 那样将多边形进行三角形化，虽然这是一个很不错的方法，并且许多人也都这么做，但是通常这样解决实在是有点太过复杂。

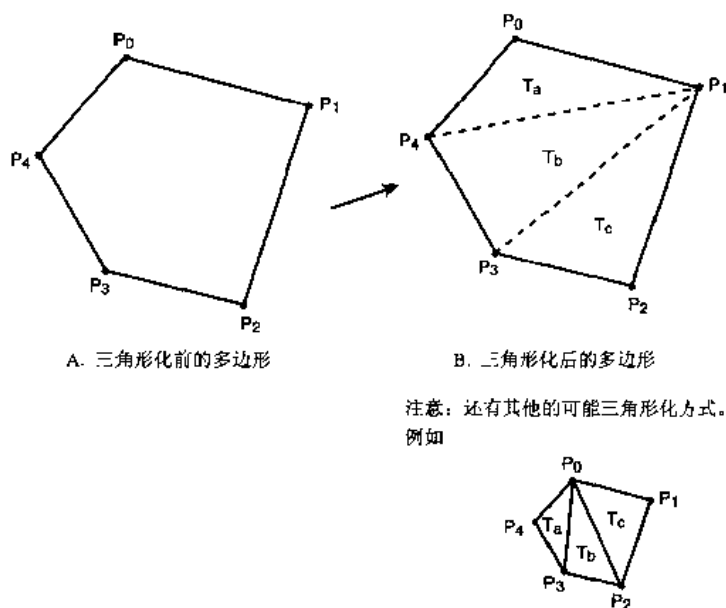


图 8.34 将一个大三角形分割

但是，如果多边形为凸多边形，它就比较简单了，有许多新的算法都可以完成这件事，但我通常使用的是非常简单，多次使用的一种方法。图 8.35 给出了五边形进行三角形化的例子。

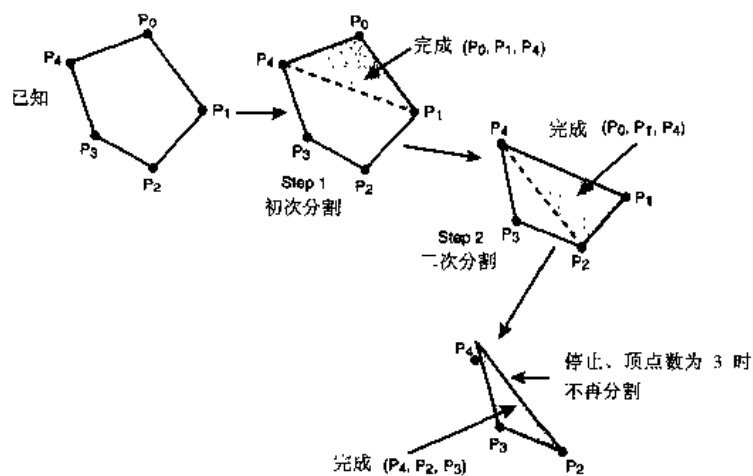


图 8.35 一种可能的三角形分割算法

请注意在图 8.35 中有几个有效的三角形。因此，为了优化三角化过程，可以使用新的或者更新的函数来进行三角形化。例如，采用面积相近的三角形进行三角形划分就不失为一个好主意，或许你可以首先采用大的三角形进行尝试，无论何种情形，它是和你的最终引擎有关的。这里给出通用的算法。

假设有一个 n (n 既可以是偶数也可以是奇数) 个顶点的多边形，顶点以顺时针或逆时针的顺序进行三角形化。

1. 如果剩余的顶点数目大于 3，到第 2 步，否则停止。
2. 取出前三个顶点组成一个三角形。
3. 去掉新建的三角形，将剩下的 $n-1$ 个顶点重复第 2 步。

基本上，算法是“剥离”三角形，再将剩下的顶点重新提交给算法，虽然没有进行任何预处理和测试，显得很乱，但确实好用。当然，一旦你将多边形变成了三角形，你可以通过光栅化管道将它们一个个发送给渲染程序。

好了，关于算法的讨论够多了，让我们来看看一种更复杂的光栅化凸多边形的方法。如果从光栅化三角形的方法来说，光栅化 n 边凸边形就非常简单了。

通过图 8.36 来看算法是如何进行工作的，一般情况下，你需要做的就是从上至下，从左至右对顶点进行排序，得到一个以逆时针方向排列的顶点阵列，然后，从最高的顶点开始，光栅化从该顶点发出的两条边（左边和右边）。当一个边到达了终点，也就是到达了第二个顶点时，重新计算光栅化的步长即 dx_left 和 dx_right ，然后再继续进行多边形的光栅化，这其实就是要做的。算法的流程图如图 8.37 所示。再次强调，有一些边界条件需要慎重考虑。例如在顶点过渡时，不要使一个边的插值计算失去同步。再次说明，图形的分割有图形分割和对象分割两种情况，让我们再讨论讨论这个问题吧。

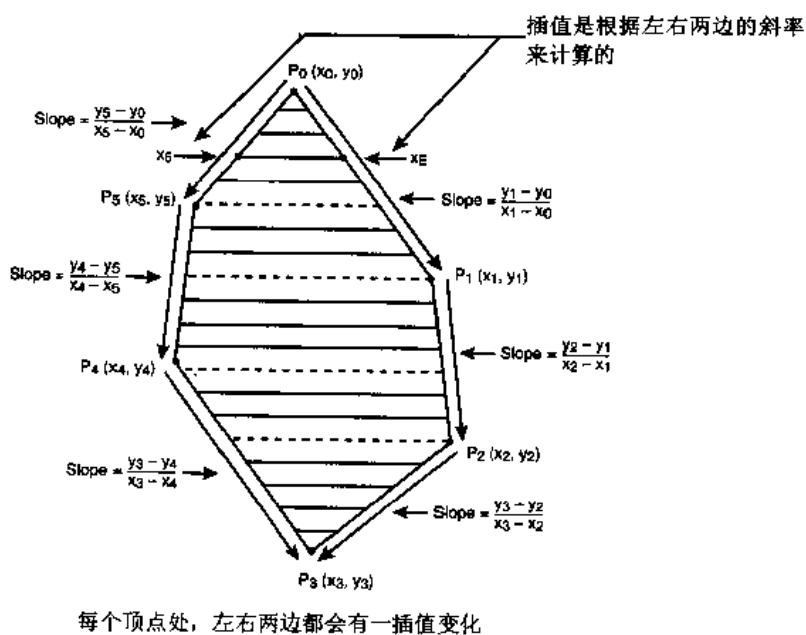


图 8.36 不使用三角形分割法来光栅化一个多边形

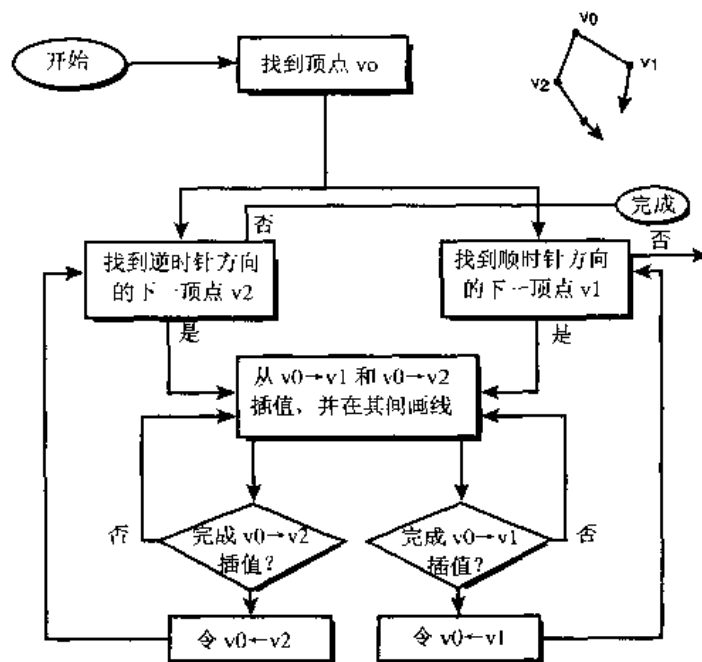


图 8.37 n 边形渲染算法流程图

当进行三角形的分割时, 你可能不愿意采用图像分割, 因为如果三个顶点都被剪掉的话, 你最后会把三角形变成一个六边形, 这很糟糕, 因为你不得不再次将一个六边形转变成三角形。但是, 因为你的新的算法适用于多边形, 谁会管它顶点增加不增加呢?

然而, 有一点你必须考虑, 就是凸多边形会不会在剪切的过程中会不会变成凹多边形? 绝对会, 但是(凡事都有但是)只有当剪切域本身是凹多边形时, 才有可能。因此, 把一个凹多边形剪切到屏幕的矩形时, 每有一个顶点落在剪切域之外时, 就至少会增加一个顶点。

当你在光栅化一个 n 多边形时, 最好的方法是以对象空间方式, 也就是在不含内部扫描线剪切代码的情况下, 以对象空间的方式进行剪切和多边形的光栅化。这就是你在这儿要使用的方法。

下面的函数采用一个标准的 `POLYGON2D_PTR`, 同时给出了帧缓冲地址和内存空间, 然后对发送的多边形进行光栅化。当然, 多边形必须为凸多边形, 且所有的顶点都必须在剪切域之内, 因为函数并不进行剪切。这里给出函数的原型。

```
Void Draw_Filled_Polygon2D(POLYGON2D_PTR poly,
                           UCHAR *vbuffer, int mempitch);
```

要画一个中心在 (320, 240) 边长为 100×100 的正方形, 需要这样做:

```
POLYGON2D square; // used to hold the square
```



```

// define points of object(must be convex)
VERTEX2DF square_verticss[4]
    ={-50, -50, 50, -50, 50, 50, -50, 50};
// initialize square
object.state =1; // turn it on
object.num_verts =4;
object.x0 =320;
object.y0 =240;
object.xv =0;
object.yv =0;
object.color =255; // white
object.vlist =new VERTEX2DF [square.num_verts];

// copy the vertices into polygon
for (int index=0; index<square.num_verts; index++)
square.vlist[index] = square_verticss[index];

//_in the main game loop
Draw_Filled_Polygon2D(&square, (UCHAR *)ddsd.lpSurface, ddsd.lPitch);

```

喔哈!感觉如何?不管怎样,我愿意给你展示函数的列表,但是它太大了。不过你可以在你的 DEMO8_9.CPIEXE 看到用该函数画一个四边形然后填充它的例子。不过,该函数不是将四边形剪切为两个三角形,而是直接对多边形进行了渲染。

技巧



通过测试对旋转对象的渲染来检测所写的渲染函数总是一个好主意,许多时候,当你检测一个光栅化函数时,你会给它一个“简单”的坐标,但是当你旋转对象时,你会得到各种值,如果光栅化函数能够完成对 360 度旋转的对象成功渲染,你的函数就工作得比较好。

多边形碰撞检测

谢天谢地,完成了前面的工作,我可以用它们进行光栅化和变换了,让我们休息一下,并谈谈与游戏相关的话题,如碰撞检测及如何决定多边形的碰撞问题。记住这点,我现在将给你关于此问题三种解决方法,通过使用这些技术,你应该能够处理你的所有多边形碰撞。

AKA 边界球/圆的接触

第一种方法是假设对象具有平均半径,通过检测半径是否重叠来检测两个多边形是否发生碰撞。这可以通过简单的距离计算来完成,如图 8.38 所示。

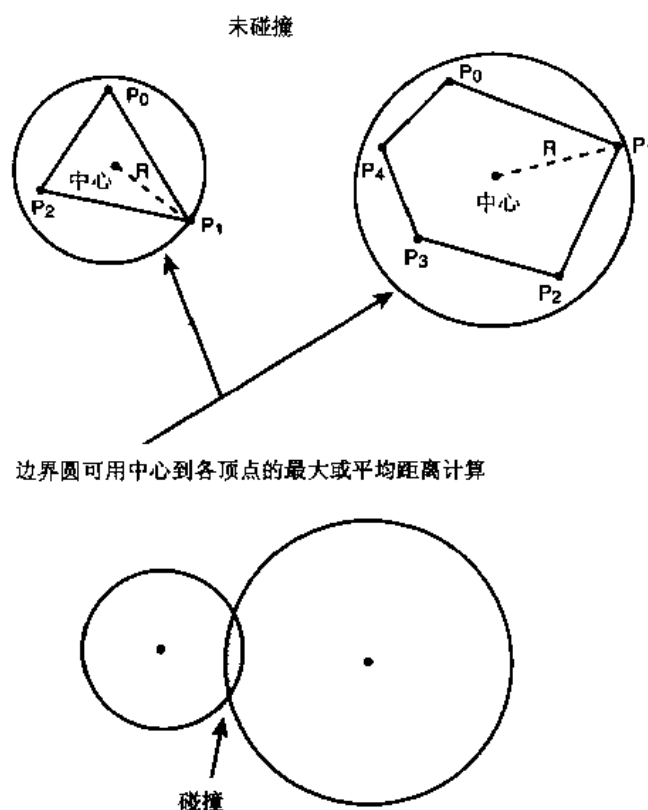


图 8.38 用边界圆测试碰撞

当然，你将多边形套进了圆形碰撞箱，采用上述方法进行监测时，会使本来没有碰撞的对象误以为碰撞。也可能使碰撞被忽略(主要要看如何选取平均半径)。

为了完成算法，首先应给多边形一个平均半径。计算平均半径的方法很多，你可以采用多边形中心到每个多边形顶点的距离平均值作为半径，也可以采用最大值或者其他的一些试验值。我通常采用平均到最远顶点距离的中点作为平均半径。无论怎样，计算可以在循环之外进行，所以不必要担心占用 CPU 的时片。但是，在实时运行时的检测确实是一个问题。

数 学

计算二维空间两点 (x_1, y_1) , (x_2, y_2) 之间的距离可以采用公式 $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ ，对于三维空间，只需要在开方项中加上 $(z_1 - z_2)^2$ 。

假设你有两个多边形，poly1 位置在 (x_1, y_1) ，poly2 位置在 (x_2, y_2) ，半径分别为 r_1 和 r_2 (不管采用何种方法)。要测试多边形是否重叠，你可以采用下面的代码。

```
// compute the distance between the center of each polygon
dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));

// test the distance
```

```

if (dist<=(r1+r2))
{
// collision!
} // end if

```

这样就可以按你想像的那样工作了，但有一个问题就是它工作起来特别慢。开方函数占据了 CPU 的大量时间，你不得不想办法进行克服。让我们从一个简单的例子开始。首先，没有必要分别计算 (x_1-x_2) 、 (y_1-y_2) 两次。你可以计算一次，再在后面利用前面计算的结果，如下面这样：

```

float dx=(x1-x2);
float dy=(y1-y2);
dist = sqrt(dx*dx+dy*dy);

```

这样做起到一点作用，但是开方花费时间是浮点数乘法的 70 倍，也就是，浮点数乘法占用 CPU 芯片大约 1~3 时片，而开方大约占用 70 时片的时间。无论那种情况，这都是无法接受的，来看看应该如何做，技巧之一是采用 Taylor/Maclaurin 公式。

数 学

Taylor/Maclaurin 公式是将复杂的函数计算简化为一些简单项之和，同时要考虑函数的导数，Maclaurin 表达式一般为：

$$f(0)+f'(0)x^1/1!+f''(0)x^2/2!+\dots+f^{(n)}(0)x^n/n!$$

 这里，'指求导，! 指阶乘，如 $3! = 6$ 。

知道上面的数学知识之后，你就可以用几次比较简单的加法运算取代 2D 或 3D 中两点 p1、p2 的距离了，这里是 2D 及 3D 情形的算法：

```

// used to compute the min and max of two expressions
#define MIN(a, b) (( < ) ? : )
#define MAX(a, b) (( > ) ? : )

int Fast_Distance_2D(int x, int y)
{
// this function computes the distance from 0,0 to x,y with 3.5% error

// first compute the absolute value of x,y
x = abs(x);
y = abs(y);

// compute the minimum of x,y
int mn = MIN(x,y);

// return the distance
return(x+y-(mn>>1)-(mn>>2)+(mn>>4));

} // end Fast_Distance_2D

```

```

////////////////////////////////////
float Fast_Distance_3D(float fx, float fy, float fz)
{
    // this function computes the distance from the origin to x,y,z

    int temp; // used for swapping
    int x,y,z; // used for algorithm

    // make sure values are all positive
    x = fabs(fx) * 1024;
    y = fabs(fy) * 1024;
    z = fabs(fz) * 1024;

    // sort values
    if (y < x) SWAP(x,y,temp)
    if (z < y) SWAP(y,z,temp)
    if (y < x) SWAP(x,y,temp)

    int dist = (z + 11*(y >> 5) + (x >> 2) );

    // compute distance with 8% error
    return((float)(dist >> 10));

} // end Fast_Distance_3D

```

每个函数的参数只不过是差值，如果用 `Fast_Distance_2D()` 计算前面的算法中的距离，你可以进行如下调用。

`Dist = Fast_Distance_2D(x1-x2, y1-y2);`

这个基于函数的新技术，只需要三次移动，四次加法，一次比较和两个绝对值运算，要快多了。

注意



要留心两种算法都是近似计算，如果需要精确值时应该小心。2D 计算有 3.5% 误差，而 3D 中是 8%。

最后，细心的读者会注意到，可以利用另外一种算法，根本不需要去求平方根。我是指如果你想知道一个对象是不是在另外一个对象 100 单位距离之内，你知道距离是： $\text{dist} = \sqrt{x^2 + y^2}$ ，但是如果对两边同时平方就得到：

$$\text{dist}^2 = x^2 + y^2$$

这里 `dist` 等 100，而 $100^2=10000$ ，因此你只需判断等式的右侧是不是小于 10000 即可，

它等价于其平方根小于 100。非常好！这种方法惟一的缺点是溢出。但是没有理由去计算实际的距离，只需比较距离的平方就可以了。

边界箱

虽然边界圆/球的算法非常直接，问题是适用对象必须近似为球形。这有时合适，有时不合适。见图 8.39，它给出了一个矩形对象，对它应用球形边界箱近似会有许多错误。所以采用近似对象本身的几何图形会更好。这种情形下，你可以采用其他边界箱（正方形或长方形）进行碰撞检测。

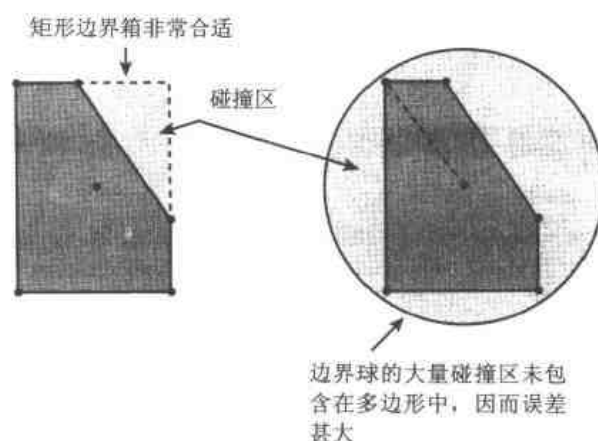


图 8.39 使用最好的边界几何方法完成工作

为多边形创建边界矩形同边界球一样，但它是需要找出四条边而不是半径。我通常将它们称为 $(\max_x, \min_x, \max_y, \min_y)$ ，它们与多边形中心有关。图 8.40 给出了创建的例子。

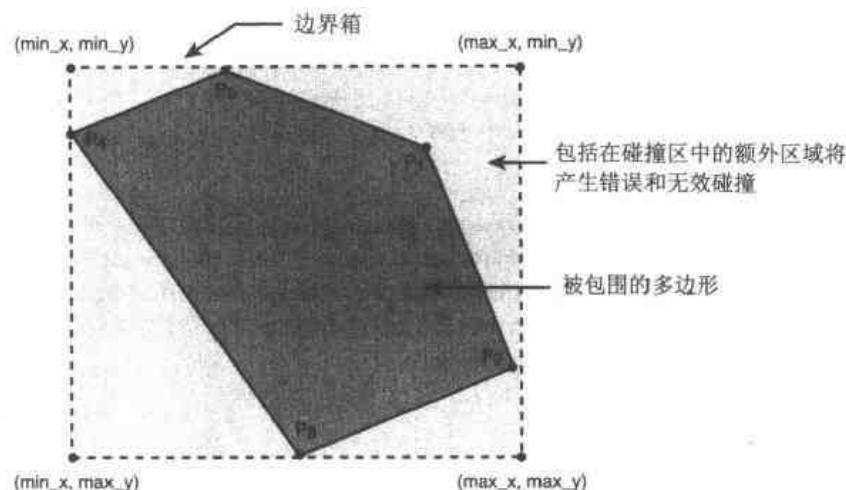


图 8.40 边界四边形方法

为了找出 (max_x, min_x, max_y, min_y) 的值, 你可以采用如下的简单算法。

1. 初始化 (max_x=0, min_x=0, max_y=0, min_y=0)。假设中心在 (0, 0)。

2. 对多边形的每个顶点, 计算 (max_x, min_x, max_y, min_y) 的 (x, y) 分量, 并作近似调整。

这里给出你标准的 POLYGON2D 结构代码。

```
int Find_Bounding_Box_Poly2D(POLYGON2D_PTR poly,
                             float &min_x, float &max_x,
                             float &min_y, float &max_y)
{
    // this function finds the bounding box of a 2D polygon
    // and returns the values in the sent vars

    // is this poly valid?
    if (poly->num_verts == 0)
        return(0);

    // initialize output vars (note they are pointers)
    // also note that the algorithm assumes local coordinates
    // that is, the poly verts are relative to 0,0
    max_x = max_y = min_x = min_y = 0;

    // process each vertex
    for (int index=0; index < poly->num_verts; index++)
    {
        // update vars - run min/max seek
        if (poly->vlist[index].x > max_x)
            max_x = poly->vlist[index].x;

        if (poly->vlist[index].x < min_x)
            min_x = poly->vlist[index].x;

        if (poly->vlist[index].y > max_y)
            max_y = poly->vlist[index].y;

        if (poly->vlist[index].y < min_y)
            min_y = poly->vlist[index].y;
    } // end for index

    // return success
    return(1);

} // end Find_Bounding_Box_Poly2D
```

注 意



函数通过了“引用”来传递参数，使用了&操作符。这与指针有点类似，不过你不能废除引用。另外，和指针不同的是&引用的是变量的别名。

你可以如下调用函数。

```
POLYGON2D poly; // assume this is initialized
Float min_x, min_y, max_x, max_y; // used to hold results
// make call
Find_Bounding_Box_Poly2D(&poly, min_x, min_y, max_x, max_y);
```

调用后建立了最大/最小矩形，并存储在(min_x, max_x, min_y, max_y)中。通过这些值加上多边形的位置(x_0, y_0)，你可以来判断两个边界箱的碰撞。当然，你可以有多种办法来作到这一点，包括测量任意一个顶点是否在另外一个边界箱的内部，或者其他更加聪明的技术。

点包含

在我给你最后讲述点是否被包含在矩形内部之前，我认为给你讲述点是否包含在一个普通的多边形之内更是一个好主意。你认为如何？显然，判断一点是否在矩形区域之内只需要如下这么做。

判断点(x_0, y_0)是否在矩形(x_1, y_1)(x_2, y_2)之内：

```
if (x0>=x1 && x0<=x2) // x-axis containment
    if (y0>=y1 && y0<=y2) // y-axis containment
    { /* point is contained */ }
```

注 意



我可以通过一个“if”语句和另外一个“&&”来连接两项，但是这些代码可以更清楚地说明线性分割的问题——也就是x, y轴可以被分别操作。

让我们看看你能否判断点是否是多边形中，如图 8.41 那样。首先你认为它是一个简单的问题。但我要提醒你并非如此。解决此问题有许多途径，但是最为直接的方法是半空间检测。如果你进行检测的多边形是一个凸多边形，你可以将每条边想像成一个无限大的平面同多边形的相交线段。每个平面将空间分割成两部分，如图 8.42 所示。

如果检测的点在每个半空间的里侧，则根据凸多边形的性质，点也在多边形的内部。因此，你现在需要找到一种二维空间判断点在哪一侧的方法。

如果你能够将直线以一定顺序进行标记，并将之转变成矢量，这就不算糟糕。之后你将每条直线想像成一个平面，利用点积算法，你可以确定点在哪一侧或者是在平面上。这

是算法的基础。

你可能对矢量、点积等等不能够很快适应，所以现在为了不把你给弄迷糊，让我们先不必忙于对点进行检测，等我们掌握了三维数学之后再来完成算法。但是你应该从几何角度进行理解解决问题的方法。如果能够进行合适的讲解，你就会发现，它的细节不过是一些数学问题。通过高水平、有机、无机的一些功能来完成数学运算。

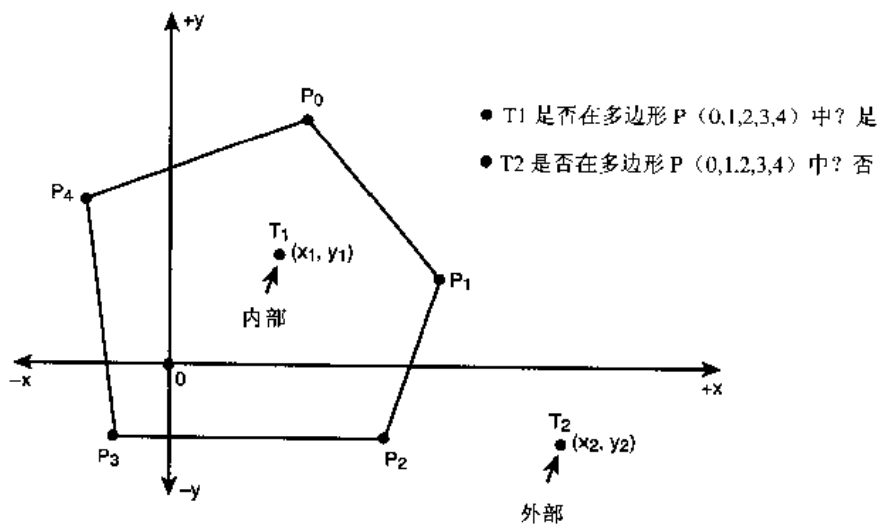


图 8.41 多边形内部点测试的设置

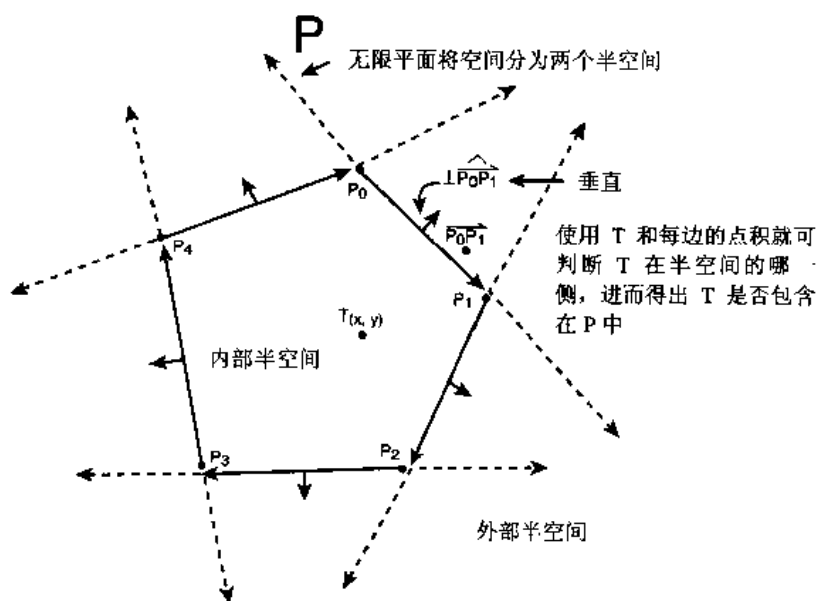


图 8.42 使用半空间帮助解决多边形内部的问题

定时与同步详解

至此，在大多数程序中，我们使用了计时函数 `sleep()`。这是你能够使用的低级技术。实际上，在你的游戏中，你有时需要锁定帧速度如 30fps。一个较好的方法是在循环开始时进行计时，记下当时的时间，最后在游戏循环结尾检测是否 30fps。也就是看看是否消耗了 30 分之一秒。如果是，显示下一帧。如果不是，等待到耗尽剩余时间为止（可以通过计算下一帧图像或者空转耗时）。

在计算机代码中，你应该像下面这样构建你的 `Game_Main()` 程序。

```
DWORD Get_Clock(void);
DWORD Start_Clock(void);
DWORD Wait_Clock(DWORD count);

int Game_Main(void *parms = NULL, int num_parms = 0)
{
    // this is called each frame

    // get the current time in milliseconds since windows
    // was started
    Get_Clock();

    // do work...

    // sync to frame rate, 30 fps in this case
    Wait_Clock(30);

} // end Game_Main .
```

如此简单吗？是的。不过这些幻想的函数到底是什么样子呢？它们是基于 Win32 的计算函数。

```
DWORD Get_Clock(void);
{
    // this function returns the current tick count

    // return time
    return(GetTickCount());
} end Get_Clock

////////////////////////////////////

DWORD Start_Clock(void);
{
```

```
// this function starts the clock, that is saves the current
// count, use in conjunction with Wait_Clock()
// sync to frame rate, 30 fps in this case

return(start_clock_count =Get_clock());
} // end Start_Clock

////////////////////////////////////

DWORD Wait_Clock(DWORD count);
{
// this function is used to wait for a specific number of clicks
// since the call to Start_Clock()

while ((Get_Clock() - start_clock_count) < count);
return(Get_clock());

} // end Wait_Clock
```

注意，基于 Win32 的函数 `GetTickCount()`，它是以字为单位返回一个毫秒时间值，记录了 Windows 运行后的时间，因此时间是相对的，不过，谁会管它是不是相对的呢？它给出了你所需要的一切。也要注意采用 `Start_Clock_Count` 全局变量来存储时间。每次调用 `Get_clock()` 函数就要修正一次它的值。在库中它是这样定义的：

```
DWORD start_clock_count = 0; // used for timing
```

C++

这里是 C++ 类的用武之地。轻松一点。

DirectX 有一个垂直空间测量函数，你可以在阴极射线管对图像着色时使用它来测量电子枪的状态。IDIRECTDRAW4 接口支持函数 `WaitForVerticalBlank()`，如下所示：

```
HRESULT WaitForVerticalBlank(DWORD dwFlags,
                             HANDLE hEvent);
```

你可以通过它来决定垂直空间的不同情况。dwFlags 控制函数的操作。hEvent 是指向 Win32 事件的句柄（高级标志）。表 8.3 给出了有效标志位的设置。

表 8.3 WaitForVerticalBlank() 的标志位设置

标 志 位	描 述
DDWAITVB_BLOCKBEGIN	当垂直空白间隔开始时返回
DDWAITVB_BLOCKEND	当垂直空白间隔结束，显示开始时返回

滚动和视角场景

我认为滚动太容易，所以从没有详细讨论过该内容。（但为了满足需要，我将页滚动写进了《Sams Teach Yourself Game Programming In 21 Days》一书，并把分层和游戏背景滚动写进了《The Black Art Of 3D Game Programming》一书。）滚动游戏有它们自己的特色，真正解释二维空间的景物滚动技术需要花费一到两章。下面我想简要地讨论一下各种景物滚动技术，然后给出演示程序。

页滚动引擎

页滚动基本上是指当玩家在屏幕上四处走动或穿过门槛时，整个屏幕随着它从一个房间到另外一个房间的移动而调整。该技术很容易完成，并且可以用多种代码来完成。看图 8.43，你可以看到一个 640×480 像素屏幕组成的 4×2 阵列。因此整个场景是 2560×960 。这种设计的着色逻辑很简单，你可以将第一幅画面调入内存，然后将相邻的画面调入 RAM 或在硬平面上虚拟。无论采用哪一种方法，移动原理都一样。随着玩家的进展，你对一些不同的边界情况进行监测（可能是屏幕的边缘），当边界条件满足时，就可以进入另外一个房间或者将玩家的人物移到别的位置。例如：如果你从左边到右边走到了屏幕的边缘，新的一页就会出现，人物出现在屏幕的右侧。当然这有点原始，但它不过是个开始。

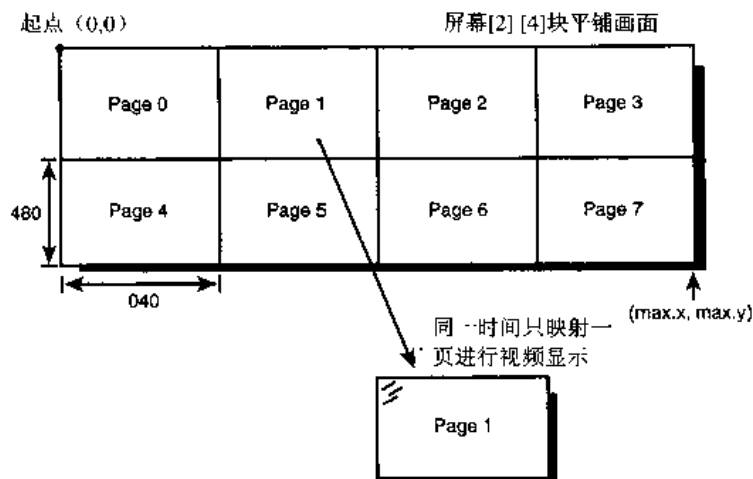


图 8.43 页滚动的全部设置

这种技术的演示程序在 CD 上名字是 DEMO8_10.EXE|CPP。它基本上是创建一个 3×1 区域，让你用箭头键移动一个小人物。当走到边缘时，画面就会调整。注意，我对屏幕使用了位图，但并不妨碍你采用矢量或混合图像。

在本章最后的文件 T3DLIB1.CPP 中的演示中，我也撒了个小小的谎，但如果你想看原

代码你可以找到。我只是比完成一个像样的演示程序需要更多的动力。

最后，一定要看看在“TERRAIN FOLLOWING”演示程序中的代码。当你通过扫描表示楼梯的不同的颜色（记得是 116 种）的索引时，游戏角色可以实现从右至左沿着楼梯移动。当扫描器发现新的颜色，人物上升一点，使其保持在地板之上。

均匀平铺显示引擎

从边上卷动游戏平台的角度来看，前面所讲的有关卷动的例子并不是真正意义上卷动。那种卷动更加平滑——不是整个屏幕一页一页地卷动，而是上下左右平滑的卷动。

用 DirectX，有多种方法可以作到这一点。你可以创建一个大的画面，而游戏只是在其中的一部分进行。如图 8.44 所示。

但是，这只是在 DirectX 6.0 或以上版本上才行，并且需要加速卡。较好的取代方法是将游戏画面分成许多块平铺显示的画面，然后用平铺显示矩阵或单元取代每个屏幕，每个单元代表在此处将要显示的位图。图 8.45 给出了它的设计原理。

例如你可以在你的 640×480 模式下建立大小为 32×32 像素的平铺显示，也就是说整个屏幕需要 $640/32 \times 480/32 = 20 \times 15$ 块平铺显示画面。或者用 64×64 平铺显示画面，这时需要 $640/64 \times 480/64 = 10 \times 7.5$ 个平铺显示画面，舍去小数部分，需要 10×7 个平铺显示（ $7 \times 64 = 448$ ，最后的 48 像素在屏幕的底部，你可以留下来安放控制按钮）。

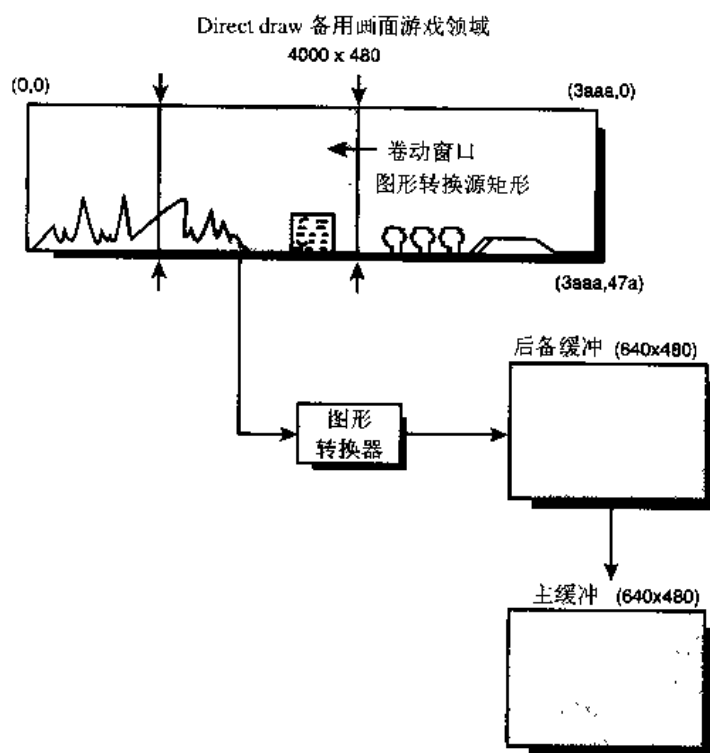


图 8.44 使用大的 DirectDraw 画面获得平滑卷动

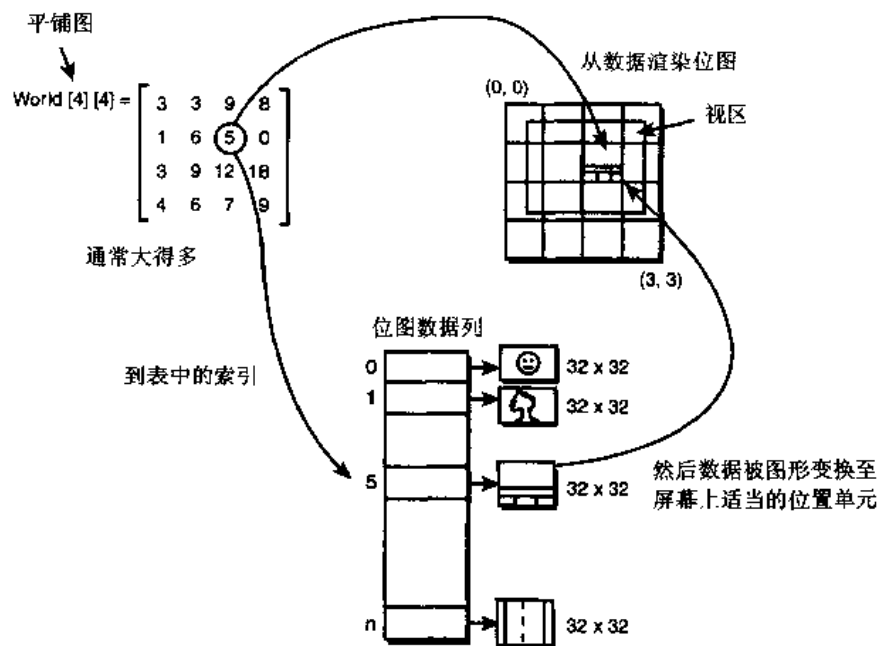


图 8.45 采用基于平铺显示数据结构表示卷动情况

完成这项工作，你需要下面的数据结构，类似一个整数排列或矩阵，也可以使结构保存位图信息（仅仅是一点或索引值），也可以保存其他必要的信息。这里给出一个创建平铺显示图像的例子：

```
typedef struct TILE_TYP
{
    int x, y; // position of tile in matrix
    int index; // index of bitmap
    int flags; // general flags for the cell
} TILE, *TILE_PTR
```

然后，为了保存屏幕的信息，可以这样：

```
typedef struct TILED_IMAGE_TYP
{
    TILE image[7][10]; // 7 rows by 10 columns
} TILED_IMAGE, *TILED_IMAGE_PTR
```

最后，给出一个 3×3 平铺显示构成的图形：

```
TILED_IMAGE world[3][3];
```

或者你可以创建一个平铺显示阵列存放 3×3 个屏幕，即 30×21 个平铺显示，如下所示：

```
typedef struct TILED_IMAGE_TYP
{
    TILE image[21][30]; // 21 rows by 30columns
} TILED_IMAGE, *TILE_IMAGE_PTR

TILED_IMAGE world;
```

注意

你可以采用任何一种方法进行数据结构的设计,但是采用的排列工作起来更加简单,因为在你滚动 10×7 个平铺显示图形后,不必再处理屏幕的图像的替换问题。

那么，如何画每一个屏幕呢？首先，你需要把你的位图装入 64×64 的阵列。你可以有一个或多个平铺显示，有些也可以是重复的，如船只、边界、水等等。图 8.46 给出了平铺显示设计的例子。

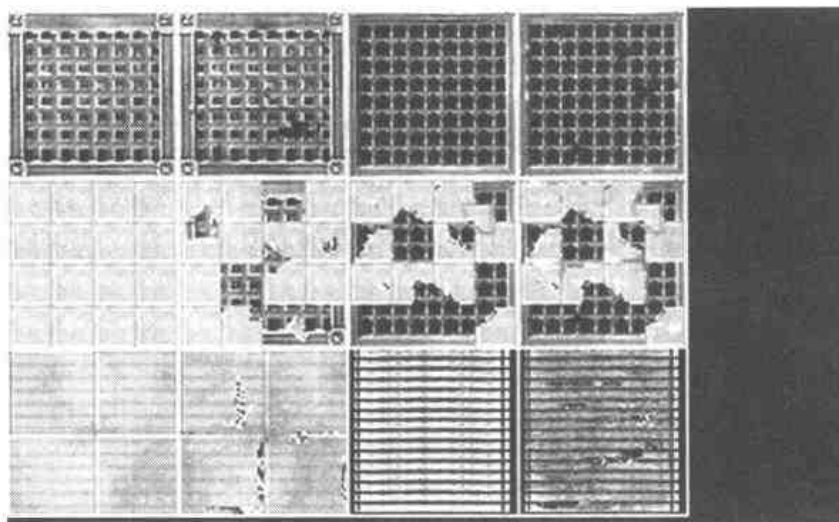


图 8.46 典型平铺显示系列位图模板

而后，你可以编写一个工具，或采用现成的 ASCII 之类的编辑器编写你的平铺显示图形。例如，你可能想用 ASCII 码和一个变换程序，0~9 代表平铺显示。如果这样，你可以只定义一个 32×21 个单元的平铺显示阵列，我是这样做的：

[illegible]

}

要建立一个玩家当前看到的一个 $m \times n$ 的视区或窗口。

你可以固定一个控制面板或者其他一些不需要变动的东西。在任何情况下，假设整个屏幕卷动大小为 640×480 ，你必须考虑以下两点：

- 边界的情况。

好吧，假设视区在屏幕的左上角(0, 0)处，如图 8.47 所示。



图 8.47 卷动平铺显示映像边界问题

这种情况下，你仅需画出从图[0][0]到图[6][9]的平铺显示。但是当视区向右移动时，就

需要考虑画出右侧进入视区的边界处的平铺显示图形，然后卷过了一个平铺显示单元后，就不需要再画图[0][0]的一整行或列平铺显示了。

所以，你在任何时候总是画一个 10×7 个平铺显示的矩形集合，这些平铺显示来自于一个或多个平铺显示图像。另外，由于你可以在 64 的倍数处的位置进行滚动，有时你只看到平铺显示的边缘，所以其中需要剪切技术。幸运的是，DirectX 将为你剪切所有的图形，所以当你画的图形只有一部分在屏幕上时，多余部分就会自动被剪切掉。因此，你最终所需做的就是决定平铺显示的着色，找出平铺显示所代表的位图，然后把它们发送给主程序。

CD 中的 DEMO811.EXE\CPP 给出了演示程序，它创建了同前面讨论的一样的绘图环境，你可以在它上面到处移动。

稀疏位图平铺显示引擎

平铺显示引擎的位移问题是需要画很多位图。有时你想编写一个滚动游戏，但是你不能有太多的图形来滚动，也并不希望所有的平铺显示都有相同的尺寸，这是一些空中射击游戏所要求的。因为许多时候这些游戏都处在一个黑色空间中。对于这种环境，你可以创建一个很大的全局图形（一般都是如此），让我们用 4×4 个屏幕（或者 40×40 ）来进行讨论。不是用平铺显示图形填充每一个子屏幕，而是将每个对象放在环境坐标系的任意位置，采用这种方法，不但可以使每个对象的大小为任意尺寸，也可以放到任何位置。

这种方案的惟一问题是其效率问题。基本上，当前视区停在任何位置时，你都需要将涉及到的所有对象找出来，以便着色。如果你在一个较小的环境中，又有较少对象，还不算太麻烦，但如果你在一个很大的环境中，有上千个对象需要测试，那麻烦就大了。

解决此问题的方法是将游戏环境分区，基本上，你可以创建一个备用的数据结构跟踪所有的对象，和它们相对分割的环境的一些单元的关系。但是，等一下，你现在不用平铺显示图像设置了吗？可以说是，也可以说不是。此时，分区可以为任意大小，和真正的屏幕尺寸没有任何关系。它们尺寸的选择和碰撞检测以及跟踪关系更密切。

如图 8.48 所示，给出了数据结构同屏幕、环境、视区的对应关系。

这只是解决这个问题的办法之一，当然还有许多其他方法。不过，如果你的对象的个数过于庞大，超出你的屏幕的范围，如 100000×100000 ，你还能否在其中到处移动呢。没问题，将所有的对象同它们的真实坐标相联接。然后，视区移动到何处就将所有对象影像或者传送到屏幕或视频缓冲中。当然，我再次提醒你要有很好的剪切程序，因为在绘制过程中，对象很可能超出显示范围。

作为稀疏卷动的例子，我编写了一个空间演示程序（当然，这不是一个高预算程序），允许你在星际间行走，同时有些星形状的对象。在这个演示程序中，没有太多的数据结构支持分区。对象本身也是随机的。在实际应用时，你当然还需要有世界地图、碰撞分区、优化渲染等等。我写的稀疏卷动演示程序名字是：DEMO8_12.CPP\EXE。它再次利用了 T3DLIB1.CPP\H 文件，在你的工程文件中要把它包含进去。

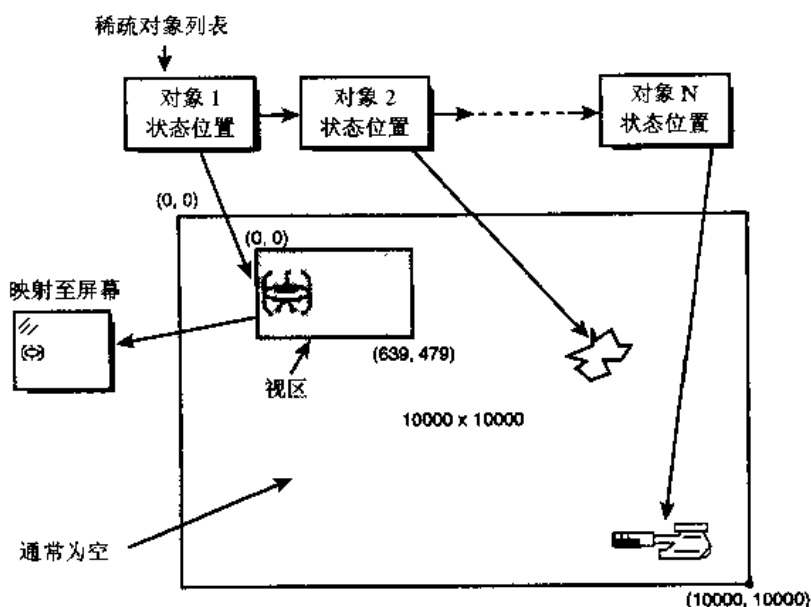


图 8.48 稀疏滚动引擎数据结构

DEMO8_12.EXE 基本上是调一些位图对象，然后再在 10×10 的屏幕上随机显示，你在环境中的行走像往常一样采用箭头键。这种滚动的动人之处在于无序地进行屏幕渲染。只需对一些看得见或者部分可见的图形进行着色。因此，对每个对象有一个剪切状态测试，如果对象根本就看不见，就无需发送给位图渲染代码段。

伪 3D 等角引擎

我已经收到很多关于这方面的邮件。我感觉应当写一本有关等角三维游戏的书。那什么是等角游戏呢？实际上就是以一定角度俯视的游戏，如 45° 。一些老的等角三维游戏有 Zaxxon、PaperBoy、Marble 和 Madness 等。

现在等角游戏又回来了，Diablo、Loaded 以及一些 RPG 战争游戏都用它。令人感兴趣的可能是它具有一些比完全采用三维更有趣的视觉效果。当然，等角游戏比完全的三维游戏要容易 10 倍。不过，你如何来做它呢？

这是游戏界的一个秘密，人们谈论得很少。我这里给出你一些信息，并描述两种编写游戏的方法。我想，这些足够你自己去完成等角引擎了。

等角三维有三种方法：

- 方法 1——基于二维单元。
- 方法 2——基于屏幕，具有一些二维和三维碰撞体系。
- 方法 3——采用全三维数学运算，带一个相机视角。

方法 1：基于完全的 2D 单元

采用方法 1，你需要在头脑中有一个视角的概念，根据它安排你的工作。一般的，你需要将所有的东西都画成矩形平铺显示，就像你在普通卷动引擎里做的那样。参见图 8.49 中的作品单元。

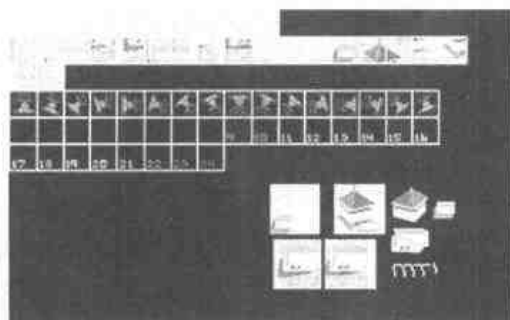


图 8.49 预先绘制的等角 3D 物体

但是，渲染时就有一定的技巧了。当进行场景绘制时，你不能以随意的顺序画对象，而必须使远的对象被近的对象所遮盖。原则是，像画家一样从后往前画。因此，如果你想有一个 45 度的俯视角，也就是说图像的顶视图倾斜了 45 度，画屏幕时就需要从顶往底画，顺序不能颠倒。

当你的游戏中有树木，人物在树木后面走动时，这点很重要。小家伙在树木后边移动会更好看。因此，你必须保证他们在绘制时出现的顺序。这个问题见图 8.50。如果你能保证在合适的时候画出人物来，即在绘制人物后边行之后，在绘制人物前边行之前，问题就十分简单。如图 8.50 所示，你不得不采用一些数学手段，理顺移动对象绘制的正确时间。

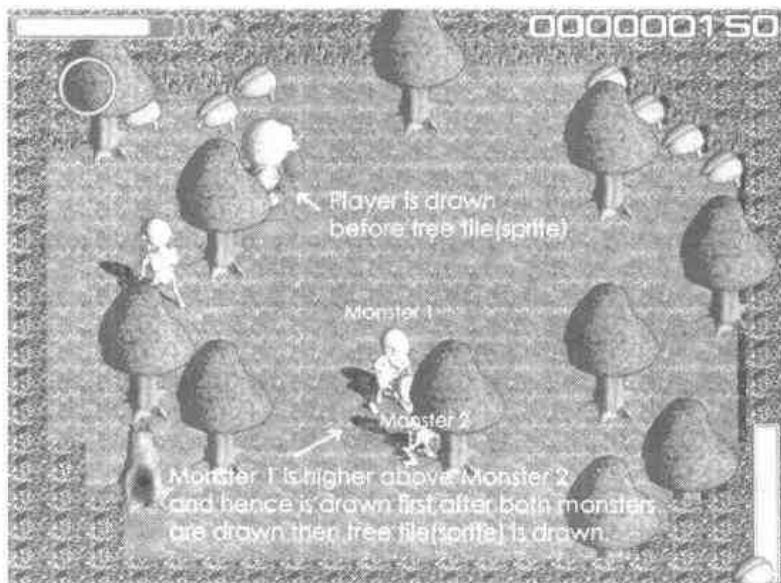


图 8.50 绘制顺序对等角 3D 渲染非常重要

当然，你可以采用具有不同平铺显示高度的等角引擎。或者换句话说，每个平铺显示可以伸展为多个平铺显示的高度。你可以简单的把多个平铺显示放在多行的位置，或者考虑高度问题，将每一行平铺显示看成不同的高度。当进行渲染时，可以在 Y 轴方向上先对比实际的行的位置更高的当前行进行渲染。

图 8.51 示意了这种情况下的演示图。在画它的时候没有什么不同；惟一的问题是加在当前行的每一块高度的起始 Y 值的定位。基本上是一个单元的高度。

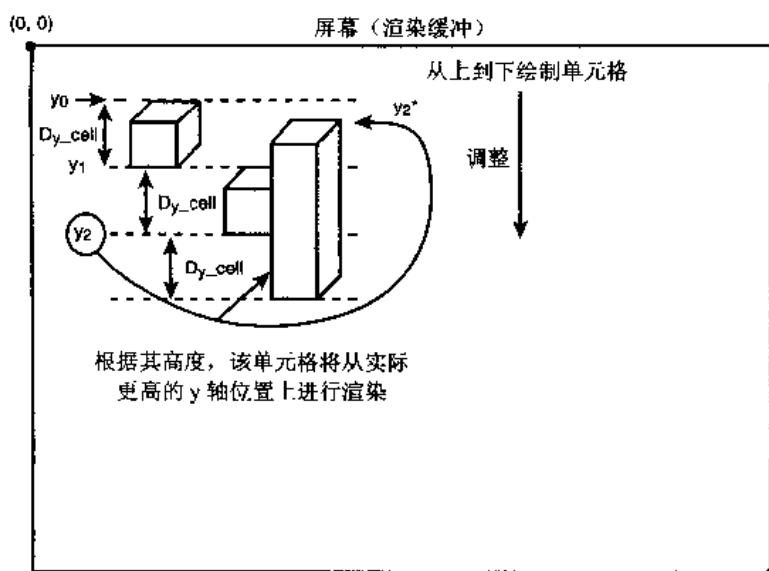


图 8.51 等角 3D 引擎中绘制大单元

现在，让我们作进一步的思考。什么是 45 度倾斜角和 45 度偏角？或者换句话说，什么是标准的 Diablo 或 Zaxxon 视角？此时，你还需要考虑在 X 轴方向的渲染顺序，你必须从左至右（或者从右至左画，要看你倾斜的方向）从上至下画。再次提醒你，你需要对对象进行 X、Y 轴两个方向的排序。

好了，想想看，就有一种办法完成它，当然，当有碰撞等情况时还会有许多细节问题，许多游戏程序员采用复杂的基于六边形或者八边形的坐标系统。然后将对象映射到此系统中，但这个技术难度非常大。

方法 2：基于全屏的 2D 和 3D 碰撞网络

全屏方法要比平铺显示方法酷。基本上你可以以你喜欢的方式绘制等角三维世界，也可以将每个屏幕的尺寸设计成你想要的大小。然后你需要一个辅助的数据结构存放碰撞信息，叠放到虚拟的二维环境中。采用这种技术，你可以将二维信息（不带有碰撞和高度数据）用额外的二维/三维信息扩大成二维或者三维，这取决于你想将程序设计多复杂。

然后，一次将背景图形画出，但是当你绘制可移动对象时，将它们剪切成额外的几何数据情况，叠加到准二维/三维图像上。见图 8.52。这里你看到的是一个二维渲染场景。

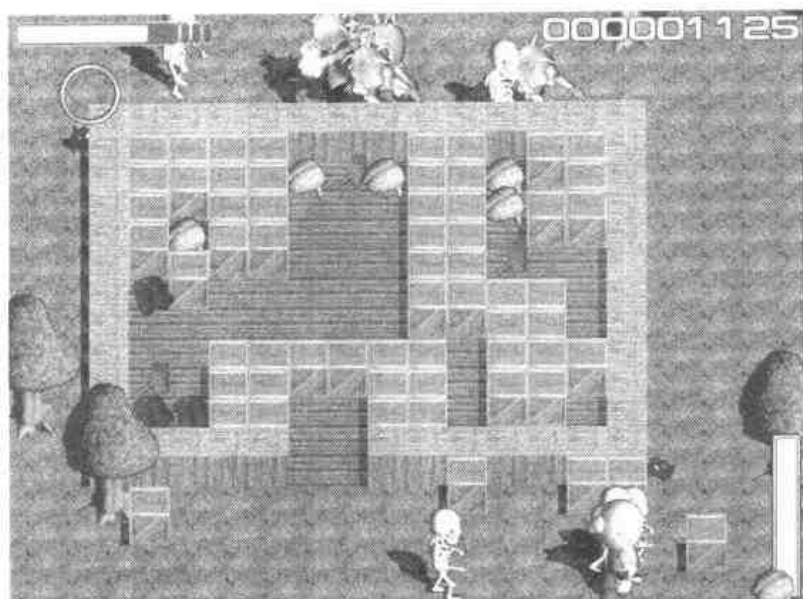


图 8.52 伪渲染等角 3D 屏幕

在图 8.53 中，你看到的是一个同样的场景，又一个多边形信息叠加在它的上面。你也可以如此进行剪切、碰撞等等。

为了完成这个效果，你需要做两件事。从三维造型中提取它，或者编写一个从全屏中提取数据的工具加入函数库。

两种方法都可以，由你来决定。但是，我建议你采用三维造型的提取程序，然后再想办法输出一些重要的几何信息。采用工具画碰撞对象需要消耗时间，而且对屏幕的任何改动都意味着有第二次改动！

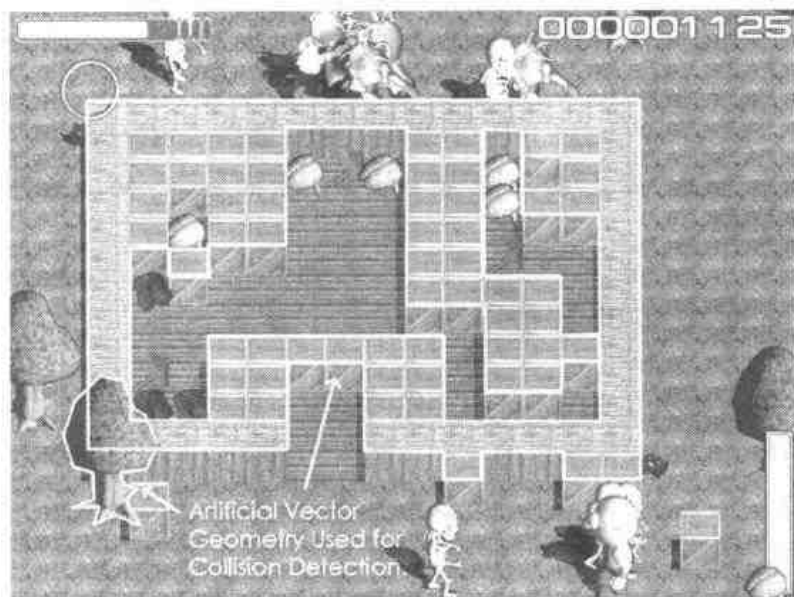


图 8.53 在伪渲染图像上多边形的碰撞

方法3：采用具有固定相机视角的全3D数学模型

这是三者之中最简单的一个，因为它没有任何技巧。基本上采用全部三维引擎。只需将相机放在等角视点处，就得到了一个等角游戏。

另外，由于你知道视点总是有一定的角度，你可以对绘制顺序和场景复杂性进行一些优化和假设。Sony 公司 Playstation I 和 II 中的许多等角游戏正是这样设计的，它们是全三维的，但是它们的视角被锁定在 45 度。

注意



就像我所说的那样，有关卷动的内容事实上是一个二维话题，并可以独立成书。所以我把另外一章放在了 CD 中，如果你觉得不够的话可以参考它。

T3DLIB1 库函数

现在来看看整个书中所用的定义、宏、数据结构、函数。另外，我还将它们放在了文件 T3DLIB1.CPP/H 中了。你可以将它们连接在你的程序中，这样可以在进一步的使用中无需从写过的大量程序中去寻找。

引擎架构

你已经将相当简单的二维引擎讨论到了图 8.54 所示的深度。基本上，它是一个二维、8 位、256 色、内部缓冲的 DirectX 引擎，可以支持任何分辨率和对主画面的剪切。

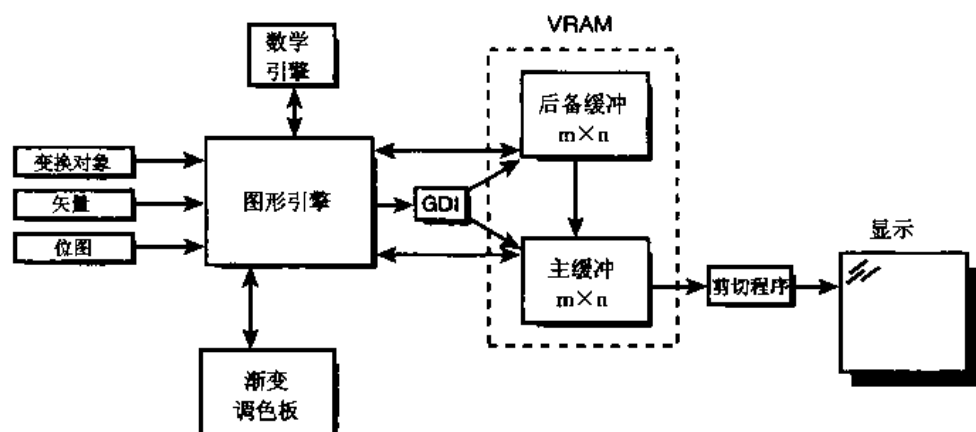


图 8.54 图形引擎的结构

但是引擎只是一个全屏幕引擎。不过它是免费的，要编写一个应用程序，将 T3DLIB1.CPPH、DDRAW.LIB(DirectX 库函数)、WINMM.LIB(Win32 多媒体库函数)加入一个基于 T3DCONSOLE.CPP(你在本书前面所编写过的)程序中就行了。

当然，你写了许多代码，可以自由改动、使用这些材料，或者按照你所想的方式随意处理。我只是想，你可以将它们收集到一个易用的文件中。当然，也可以用它得到许多 16 位程序的变换，我想把这种变换留给你自己来完成，因为我已给你详细讨论了 16/24 位方式。

基本定义

引擎有一个头文件 T3DLIB1.H。其中有许多引擎使用的#define 语句。这里给出它们的注释：

```
// DEFINES ////////////////////////////////////////

// default screen size
#define SCREEN_WIDTH      640 // size of screen
#define SCREEN_HEIGHT     480
#define SCREEN_BPP        8   // bits per pixel
#define MAX_COLORS_PALETTE 256

// bitmap defines
#define BITMAP_ID          0x4D42 // universal id for a bitmap
#define BITMAP_STATE_DEAD  0
#define BITMAP_STATE_ALIVE 1
#define BITMAP_STATE_DYING 2
#define BITMAP_ATTR_LOADED 128

#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS  1

// defines for BOBs
#define BOB_STATE_DEAD      0 // this is a dead bob
#define BOB_STATE_ALIVE     1 // this is a live bob
#define BOB_STATE_DYING     2 // this bob is dying
#define BOB_STATE_ANIM_DONE 1 // done animation state
#define MAX_BOB_FRAMES      64 // maximum number of bob frames
#define MAX_BOB_ANIMATIONS  16 // maximum number of
                               // animation sequesces

#define BOB_ATTR_SINGLE_FRAME 1 // bob has single frame
#define BOB_ATTR_MULTI_FRAME  2 // bob has multiple frames
#define BOB_ATTR_MULTI_ANIM   4 // bob has multiple animations
#define BOB_ATTR_ANIM_ONE_SHOT 8 // bob will perform
                               // the animation once
#define BOB_ATTR_VISIBLE      16 // bob is visible
```

```

#define BOB_ATTR_BOUNCE      32 // bob bounces off edges
#define BOB_ATTR_WRAPAROUND  64 // bob wraps around edges
#define BOB_ATTR_LOADED      128 // the bob has been loaded
#define BOB_ATTR_CLONE       256 // the bob is a clone

// screen transition commands
#define SCREEN_DARKNESS 0 // fade to black
#define SCREEN_WHITENESS 1 // fade to white
#define SCREEN_SWIPE_X 2 // do a horizontal swipe
#define SCREEN_SWIPE_Y 3 // do a vertical swipe
#define SCREEN_DISOLVE 4 // a pixel dissolve
#define SCREEN_SCRUNCH 5 // a square compression
#define SCREEN_BLUENESS 6 // fade to blue
#define SCREEN_REDNESS 7 // fade to red
#define SCREEN_GREENNESS 8 // fade to green

// defines for Blink_Colors
#define BLINKER_ADD 0 // add a light to database
#define BLINKER_DELETE 1 // delete a light from database
#define BLINKER_UPDATE 2 // update a light
#define BLINKER_RUN 3 // run normal

// fixed point mathematics constants
#define FIXP16_SHIFT 16
#define FIXP16_MAG 65536
#define FIXP16_DP_MASK 0x0000ffff
#define FIXP16_WP_MASK 0xffff0000
#define FIXP16_ROUND_UP 0x00008000

```

你已经在一些地方看到过它们。

运行宏

接着是一些宏，你也已经在一些地方看到过它们。这里是它们的集合。

```

// MACROS ////////////////////////////////////////

// these read the keyboard asynchronously
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// this builds a 16 bit color value in 5.5.5 format (1-bit alpha mode)
#define _RGB16BIT555(r,g,b) ((b%32) + ((g%32) << 5) + ((r%32) << 10))

// this builds a 16 bit color value in 5.6.5 format (green dominate mode)
#define _RGB16BIT565(r,g,b) ((b%32) + ((g%64) << 6) + ((r%32) << 11))

// bit manipulation macros

```

```
#define SET_BIT(word,bit_flag) ((word)=((word) | (bit_flag)))
#define RESET_BIT(word,bit_flag) ((word)=((word) & (~bit_flag)))

// initializes a direct draw struct, basically zeros
// it and sets the dwSize field
#define DDRAW_INIT_STRUCT(ddstruct)
{ memset(&ddstruct,0,sizeof(ddstruct));
ddstruct.dwSize=sizeof(ddstruct); }

// used to compute the min and max of two expressions
#define MIN(a, b) (( < ) ? : )
#define MAX(a, b) (( > ) ? : )

// used for swapping algorithm
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// some math macros
#define DEG_TO_RAD(ang) ((ang)*PI/180)
#define RAD_TO_DEG(rads) ((rads)*180/PI)
```

数据结构和类型

下面的代码包括引擎使用的数据结构和类型。我将它们列出，但是我警告你，其中有一对你没有见过的和 BOB(Billter Object Engine)有关的内容。为了内容的连续性，让我们马上来看一看：

```
// basic unsigned types
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char UCHAR;
typedef unsigned char BYTE;

// container structure for bitmaps .BMP file
typedef struct BITMAP_FILE_TAG
{
    BITMAPFILEHEADER bitmapfileheader; // this contains the
                                        // bitmapfile header
    BITMAPINFOHEADER bitmapinfoheader; // this is all the info
                                        // including the palette
    PALETTEENTRY palette[256]; // we will store
                                // the palette here
    UCHAR *buffer; // this is a pointer
                  // to the data

} BITMAP_FILE, *BITMAP_FILE_PTR;
```



```
// the blitter object structure BOB
typedef struct BOB_TYP
{
    int state;           // the state of the object (general)
    int anim_state;      // an animation state variable, up to you
    int attr;            // attributes pertaining
                        // to the object (general)
    int x,y;             // position bitmap will be displayed at
    int xv,yv;           // velocity of object
    int width, height;   // the width and height of the bob
    int width_fill;      // internal, used to force 8*x wide surfaces
    int counter_1;       // general counters
    int counter_2;
    int max_count_1;     // general threshold values;
    int max_count_2;
    int varsI[16];       // stack of 16 integers
    float varsF[16];     // stack of 16 floats
    int curr_frame;      // current animation frame
    int num_frames;      // total number of animation frames
    int curr_animation;  // index of current animation
    int anim_counter;    // used to time animation transitions
    int anim_index;      // animation element index
    int anim_count_max;  // number of cycles before animation
    int *animations[MAX_BOB_ANIMATIONS]; // animation sequences

    LPDIRECTDRAWSURFACE4 images[MAX_BOB_FRAMES]; // the bitmap images
                                                // DD surfaces

} BOB, *BOB_PTR;

// the simple bitmap image
typedef struct BITMAP_IMAGE_TYP
{
    int state;           // state of bitmap
    int attr;            // attributes of bitmap
    int x,y;             // position of bitmap
    int width, height;   // size of bitmap
    int num_bytes;       // total bytes of bitmap
    UCHAR *buffer;       // pixels of bitmap

} BITMAP_IMAGE, *BITMAP_IMAGE_PTR;

// blinking light structure
typedef struct BLINKER_TYP
{
    // user sets these
    int color_index;     // index of color to blink
    PALETTEENTRY on_color; // RGB value of "on" color
    PALETTEENTRY off_color; // RGB value of "off" color
}
```

```

    int on_time;           // number of frames to keep "on"
    int off_time;          // number of frames to keep "off"

    // internal member
    int counter;           // counter for state transitions
    int state;             // state of light, -1 off, 1 on, 0 dead
} BLINKER, *BLINKER_PTR;

// a 2D vertex
typedef struct VERTEX2DI_TYP
{
    int x,y; // the vertex
} VERTEX2DI, *VERTEX2DI_PTR;

// a 2D vertex
typedef struct VERTEX2DF_TYP
{
    float x,y; // the vertex
} VERTEX2DF, *VERTEX2DF_PTR;

// a 2D polygon
typedef struct POLYGON2D_TYP
{
    int state;             // state of polygon
    int num_verts;         // number of vertices
    int x0,y0;             // position of center of polygon
    int xv,yv;             // initial velocity
    DWORD color;           // could be index or PALETTEENTRY
    VERTEX2DF *vlist;      // pointer to vertex list

} POLYGON2D, *POLYGON2D_PTR;

// matrix defines
typedef struct MATRIX3X3_TYP
{
    float M[3][3]; // data storage
} MATRIX3X3, *MATRIX3X3_PTR;

typedef struct MATRIX1X3_TYP
{
    float M[3]; // data storage
} MATRIX1X3, *MATRIX1X3_PTR;

typedef struct MATRIX3X2_TYP
{
    float M[3][2]; // data storage
} MATRIX3X2, *MATRIX3X2_PTR;

```

```
typedef struct MATRIX1X2_TYP
{
    float M[2]; // data storage
} MATRIX1X2, *MATRIX1X2_PTR;
```

还不错，没有什么新东西，只是一些基本类型、所有的图形内容、多边形支持和一点矩阵数学。

全局控制

你知道，我喜欢用全局变量，因为它们非常之快。另外，你应该感谢许多系统级的变量(在二维/三维引擎中有好多)。所以这里给出引擎所用的全局变量，你又再次见到它们，不过这次还有注释，来看看：

```
extern FILE *fp_error; // general error file

// notice that interface 4.0 is used on a number of interfaces
extern LPDIRECTDRAW4 lpdd; // dd object
extern LPDIRECTDRAWSURFACE4 lpddsprimary; // dd primary surface
extern LPDIRECTDRAWSURFACE4 lpddsback; // dd back surface
extern LPDIRECTDRAWPALETTE lpddpal; // a pointer to the
// created dd palette
extern LPDIRECTDRAWCLIPPER lpddclipper; // dd clipper
extern PALETTEENTRY palette[256]; // color palette
extern PALETTEENTRY save_palette[256]; // used to save palettes
extern DDSURFACEDESC2 ddsd; // a direct draw surface
// description struct
extern DDBLTFX ddbltfx; // used to fill
extern DDSCAPS2 ddscaps; // a direct draw surface
// capabilities struct
extern HRESULT ddrval; // result back from dd calls
extern UCHAR *primary_buffer; // primary video buffer
extern UCHAR *back_buffer; // secondary back buffer
extern int primary_lpitch; // memory line pitch
extern int back_lpitch; // memory line pitch
extern BITMAP_FILE bitmap8bit; // a 8 bit bitmap file
extern BITMAP_FILE bitmap16bit; // a 16 bit bitmap file
extern BITMAP_FILE bitmap24bit; // a 24 bit bitmap file

extern DWORD start_clock_count; // used for timing
extern int windowed_mode; // tracks if dd is
// windowed or not

// these defined the general clipping rectangle for software clipping
```

```
extern int min_clip_x,           // clipping rectangle
          max_clip_x,
          min_clip_y,
          max_clip_y;

// these are overwritten globally by DD_Init()
extern int screen_width,        // width of screen
          screen_height,        // height of screen
          screen_bpp;           // bits per pixel

extern int window_client_x0;    // used to track the starting
                                // (x,y) client area
extern int window_client_y0;    // for windowed mode directdraw operations

// storage for our lookup tables
extern float cos_look[360];
extern float sin_look[360];
```

DirectDraw 接口

既然已经看到了支持数据，让我们来看看你写的 DirectDraw 支持函数。令人惊奇的是，你已经做了许多工作，但是你还没有意识到。DirectDraw 具有双缓冲（有一个后备缓冲），支持 256 颜色，有调色板、投影调色板、剪切区，具有写主缓冲和辅助缓冲的能力，当然，也可以进行页剪切。让我们浏览一下每个函数。

函数原型:

```
int Ddraw_Init(int width, // width of display
               int height, // height of display
               int bpp); // bit per pixel
```

目的:

DDraw_Init()启动并初始化 DirectDraw。你可以发送任何分辨率和颜色深度，成功则返回 TRUE。

例子:

```
// put the system into 800×600 with 256 colors
DDraw_Init(800, 600, 8);
```

函数原型:

```
int DDraw_Shutdown(void);
```

目的:

DDraw_Shutdown()关闭 DirectDraw, 释放所有界面。

例子:

```
// in your system shutdown code you might put
DDraw_Shutdown(void);
```

函数原型:

```
LPDIRECTDRAWCLIPPER
    Ddraw_Attach_Clipper(
        LPDIRECTDRAWCLIPPER4 lpdds, // surface to attach to
        int num_rects, // number of rects
        LPRECT clip_list); // pointer to rects
```

目的:

DDraw_Attch_Clipper()连接一个剪切板给画面 (一般是备用缓冲), 另外, 必须将剪切矩形的数目和指针发送给 RECT 序列, 如果成功则返回 TRUE。

例子:

```
//creates a clipping region the size of the screen
RECT clip_zone = {0,0,SCREEN_WIDTH-1,SCREEN_HEIGHT-1};
Ddraw_Attach_Clipper(lpddsback,1,&clip_zone);
```

函数原型:

```
LPDIRECTDRAWSURFACE4
    DDraw_Create_Surface(int width, // width of surface
                        Int height, // height of surface
                        Int mem_flags); // control flags
```

目的:

DDraw_Creat_Surface()用来在系统内存、VRAM 或 AGP 内存中创建一个画面外 DirectDraw 图形。默认情况下是 DDSCAPS_OFFSCREENPLAIN。任何其他控制标志都与默认值进行“或”运算。它们是标准的 DirectDraw 的 DDSCAP* 标志, 如:DDSCAPS_SYSTEMMEMORY 或者 DDSCAPS_VIDEO_MEMORY, 分别对应系统内存和 VRAM。如果函数成功, 则返回指向图形的指针。否则返回 NULL。

例子:

```
// let's create a 64X64 surface in VRAM
LPDIRECTDRAWSURFACE4 image =
    DDraw_Create_Surface(64,64, DDSCAPS_VIDEOMEMORY);
```

函数原型:

```
Int DDraw_Flip(void);
```

目的:

DDraw_Flip()简单地利用第二个画面剪切第一主画面，调用等待剪切发生，不能够马上返回。调用成功则返回 **TRUE**。

例子:

```
//flip em baby
DDraw_Flip();
```

函数原型:

```
Int DDraw_Wait_For_Vsync(void);
```

目的:

在下一个空闲周期开始（当光栅化到屏幕的底部时）前该函数处于等待状态。成功则返回 **TRUE**，否则返回 **FALSE**。

例子:

```
// wait 1/70th of sec
DDraw_Wait_For_Vsync();
```

函数原型:

```
Int DDraw_Fill_Surface(LPDIRECTDRAW_SURFACE4 lpdds, int color);
```

目的:

DDraw_Fill_Surface()用来采用一种颜色填充画面。颜色必须采用画面的颜色深度，如单字节 256 色模式或者 RGB 真彩色模式。成功则返回 **TRUE**。

例子:

```
//fill the primary surface with color 0
DDraw_Fill_Surface(lpddsprimary,0);
```

函数原型:

```
UCHAR *DDraw_Lock_Surface(LPDIRECTDRAW_SURFACE4 lpdds, int *lpitch);
```

目的:

DDraw_Lock_Surface()锁定发送的画面（如果可能），返回指向画面的 **UCHAR** 指针，

用画面内存的间距调整变量 `lpitch`。当画面被锁定时，你可以往上面绘制像素点或进行其他操作，但是，图形变换器会被阻塞，记住要进行 ASAP 解锁。另外，对画面解锁之后，内存指针和间距变成无效，不能够再次被使用。如果成功则 `DDraw_Lock_Surface()` 返回画面内存的非空地址。否则返回 `NULL`。

例子:

```
// holds the memory pitch
int lpitch = 0;

// let's lock the little 64X64 image we made
UCHAR *memory = DDraw_Lock_Surface(image, &lpitch);
```

函数原型:

```
Int DDraw_Unlock_Surface(LPDIRECTDRAWSURFACE4 lpdds);
```

目的:

`DDraw_Unlock_Surface()` 在锁定之前解锁一个画面。只需要将指针传递给该画面。如果成功则返回 `TRUE`。

例子:

```
// unlock the image surface
DDraw_Unlock_Surface(image);
```

函数原型:

```
UCHAR *DDraw_Lock_Back_Surface(void);
UCHAR *DDraw_Lock_Primary_Surface(void);
```

目的:

这两个函数用来锁定主画面和备用画面。但是，多数时候你只是对锁定双缓冲系统中的备用画面感兴趣，但是，如果需要，具有锁定主画面的能力。如果调用 `Ddraw_Lock_Primary_Surface()`，下面的全局变量就有效:

```
Extern UCHAR *primary_buffer;    // primary video buffer
Extern int    primary_lpitch;    // memory line pitch
```

之后你就可以随心所欲操纵画面内存;但是图形变换器将被锁定。调用函数 `Ddraw_Lock_Back_Surface()` 将锁定后备缓冲画面，并使得下面的两个全局变量激活。

```
Extern UCHAR *back_buffer;      // secondary back buffer
Extern int    back_lpitch;      // memory line pitch
```

注 意



你自己不要做这些全局变量的改动。它们用来追踪锁定函数的状态。改变它们可能会使系统崩溃。

例子:

```
// let lock the primary surface and write a pixel to the
// upper left hand corner
DDraw_Lock_Primary();

Primary_buffer[0] = 100;
```

函数原型:

```
Int DDraw_Unlock_Primary_Surface(void);
Int DDraw_Unlock_Back_Surface(void);
```

目的:

函数被用来解锁主画面或者后备缓冲画面。如果想要解锁没有锁定的画面，将不起作用，成功则返回 TRUE。

例子:

```
// unlock the secondary back buffer
DDraw_Unlock_Back();
```

二维多边形函数

下面的函数组成二维多边形系统。这些函数决不意味着先进、快速、高效，它们只是按你指定的工作做。有更好的方法替代下面的内容，这也是你学习本书的原因。

函数原型:

```
Void Draw_Triangle_2D(int x1, int y1, // triangle vertices
    Int x2,int y2,
    Int x3,int y3,
    Int color, //8-bit color index
    UCHAR *dest_buffer, // destination buffer
    Int mempitch); // memory pitch

// fixed point high speed version, slightly less accurate
void Draw_TriangleFP_2D(int x1, int y1,
```



```

    Int x2,int y2,
    Int x3,int y3,
    Int color,
    UCHAR *dest_buffer,
    Int mempitch);

```

目的:

`Draw_Triangle_2D()`函数在给定的内存缓冲用发送过来的颜色绘制填充三角形。三角形将被剪切到全局变量的当前剪切区,而不是 `DirectDraw` 的剪切板。这是因为函数采用了软件而不是硬件画线。注意: `Draw_TriangleFP_2D()`也做同样的工作,但是,因其内部使用混合坐标点,因此速度快了一点,不过精度稍微欠佳。两个函数没有返回值。

例子:

```

// draw a triangle (100,10) (150,50) (50,60)
// with color index 50 in the back buffer surface
Draw_Triangle_2D(100,10,150,50,50,60,
    50, // color index 50
    back_buffer,
    back_lpitch);

```

函数原型:

```

inline void Draw_QuadFP_2D(int x0,int y0, // vertices
    int x1,int y1,
    Int x2,int y2,
    Int x3,int y3,
    Int color, //8-bit color index
    UCHAR *dest_buffer, // destination video buffer
    Int mempitch); // memory pitch of buffer

```

目的:

`Draw_QuadFP_2D()`函数绘制由两个三角形组成的四边形。没有返回值。

例子:

```

// draw a quadrilateral, note vertices must be ordered
// either in cw or ccw order
Draw_QuadFP_2D(0,0, 10,0,15,20,5,25,
    100,
    back_buffer, back_lpitch);

```

函数原型:

```

void Draw_Filled_Polygon2D(
    POLYGON2D_PTR poly, // poly to render
    UCHAR *vbuffer, // video buffer
    Int mempitch); // memory pitch

```

目的:

`Draw_Filled_Polygon2D()`绘制一般的填充的 n 边形。函数使多边形着色, 设一个指针指向视频缓冲以及间距等等。注意:函数相对于多边形的(x0, y0)点着色, 所以要确保它们被初始化。无返回值。

例子:

```
// draw a polygon in the primary buffer
Draw_Filled_Polygon2D (&poly,
                      Primary_buffer,
                      Primary_lpitch);
```

函数原型:

```
Int Translate_Polygon2D(
    POLYGON2D_PTR poly, // poly to translate
    Int dx, int dy); // translation factors
```

目的:

`Translate_Polygon2D()`传送给定多边形的原点(x0, y0)。注意:该函数并不传输或者更改组成多边形的矢量。成功则返回 TRUE。

例子:

```
// translate polygon 10,-5
Translate_Polygon2D(&poly, 10, -5);
```

函数原型:

```
Int Rotate_Polygon2D(
    POLYGON2D_PTR poly, // poly to translate
    Int theta); // angle 0_359
```

目的:

`Rotate_Polygon2D()`函数沿逆时针方向围绕起原点旋转给定的多边形。角度必须是 0~359 的整数。成功则返回 TRUE。

例子:

```
// rotate polygon 10 degrees
Rotate_Polygon2d(&poly, 10);
```

函数原型:

```
Int Scale_Polygon2D(POLYGON2D_PTR poly, // poly to scale
                    Float sx, float sy ); // scale factors
```

目的:

Scale_polygon2D()以 sx、sy 为缩放因子分别在 x-、y-轴方向缩放给定的多边形。无返回值。

例子:

```
// scale the poly equally 2x
Scale_polygon2D(&poly, 2, 2);
```

2D 图形的基本函数

这套函数把各种东西都包含了一点进来；它是各种图形函数的大杂烩。没有你没有见过的东西——至少我这么认为。

函数原型:

```
Int Draw_Clip_Line(int x0,int y0, // starting point
                  Int x1,int y1, // ending point
                  UCHAR color, // 8-bit color
                  UCHAR *dest_buffer, // video buffer
                  Int lpitch); // memory pitch
```

目的:

Draw_Clip_Line()剪切发送的直线到当前的剪切矩形，然后在发送的缓冲里面画线，成功则返回 TRUE。

例子:

```
// draw a line in the back buffer from (10,10) to (100, 200)
Draw_Clip_Line(10,10,100,200,
               5, // color 5
               back_buffer,
               back_lpitch);
```

函数原型:

```
int clip_Line( int &x1, int &y1, // starting point
               int &x2, int &y2); // ending point
```

目的:

Clip_Line()大部分在内部使用，但是可以调用它剪切发送的直线到当前矩形剪切区。注意:该函数改变直线的端点，如果不想使用这个附加作用，需要事先保存它们。函数也不能够绘制任何东西，它只是剪切端点。成功则返回 TRUE。

例子:

```
// clip the line defined by x1,y1 to x2,y2
Clip_Line(x1,y1,x2,y2);
```

函数原型:

```
Int Draw_Line( int x0, int y0, //starting point
               int x1, int y1, //ending point
               UCHAR color,    // 8-bit color index
               UCHAR *vb_start, // video buffer
               int lpitch);    // memory pitch
```

目的:

Draw_Line()绘制直线，没有进行任何剪切。所以，要确保端点在显示画面的有效坐标之内。因为不必剪切，所以该函数比剪切版本稍快。成功则返回 **TRUE**。

例子:

```
// draw a line in the back buffer from(10,10)to (100,200)
Draw_Line(10,10,100,200,
          5, // color 5
          back_buffer,
          back_lpitch);
```

函数原型:

```
inline int Draw_Pixel(int x, int y, // position of pixel
                     int color, // 8-bit color
                     UCHAR *video_buffer, // gee hmm?
                     Int lpitch); // memory pitch
```

目的:

Draw_Pixel()在显示画面内存绘制一个像素点。多数时候，你不会基于像素创建对象，因为调用本身花费的时间比绘制像素的时间还要多。但是，如果速度不成问题，函数可以做这个工作。成功返回 **TRUE**。

例子:

```
// draw a pixel in the center of the 640X480 screen
Draw_Pixel(320, 240, 100, back_buffer, back_lpitch);
```

函数原型:

```
int Draw_Rectangle(int x1, int y1, // upper left corner
                  int x2, int y2, // lower right corner
                  int color, // 8-bit color
                  LPDIRECTDRAWSURFACE lpdds); // dd surface
```

目的:

`Draw_Rectangle()`函数在发送的 `DirectDraw` 画面绘制矩形。注意，画面在调用函数时，必须被解锁。另外，函数使用图形变换器，所以非常快。成功则返回 `TRUE`。

例子:

```
// fill the screen using the blitter
Draw_Rectangle(0, 0, 639, 479, 0, lpddsback);
```

函数原型:

```
void Hline(int x1, int x2, // start and end x points
           int y, // row to draw on
           int color, // 8-bit color
           UCHAR *vbuffer, // video buffer
           int lpitch); // memory pitch
```

目的:

`Hline()`函数绘制水平线比普通直线绘制函数快很多，无返回值。

例子:

```
// draw a fast line from 10, 100 to 100, 100
Hline(10, 100, 100,
      20, back_buffer, back_lpitch);
```

函数原型:

```
void Vline(int y1, int y2, // start and end y points
           int x, // row to draw on
           int color, // 8-bit color
           UCHAR *vbuffer, // video buffer
           int lpitch); // memory pitch
```

目的:

`Vline()`函数用于快速绘制垂直直线。它没有 `Hline()`快，但是比 `Draw_Line()`快。所以，当你知道一个直线总是垂直时，不妨用它来绘制。没有返回值。

例子:

```
// draw a fast line from 320, 0 to 320,479
VLine(0, 479, 320, 54,
      primary_buffer,
      primary_lpitch);
```

函数原型:

```
void Screen_Transitions(int effect, // screen transition
                        UCHAR *vbuffer, // video buffer
                        int lpitch); // memory pitch
```

目的:

Screen_Transition()执行各种内存中的屏幕转换操作, 这些转换操作列在前一个头信息中。注意, 转换是具有破坏性的, 如果在转换后还要用到它们, 就要保存原来的图像或者调色板。无返回值。

例子:

```
// draw a fast line from 320, 0 to 320,479
VLine(0, 479, 320, 54,
      primary_buffer,
      primary_lpitch);
```

函数原型:

```
Int Draw_Text_GDI(char *text, // null terminated string
                  Int x, int y, // position
                  COLORREF color, // general RGB color
                  LPDIRECTDRAWSURFACE4 lpdds); // dd surface

Int Draw_Text_GDI(char *text, // null terminated string
                  Int x, int y, // position
                  Int color, // 8-bit color index
                  LPDIRECTDRAWSURFACE4 lpdds); // dd surface
```

目的:

Draw_Text_GDI()用想要的颜色和位置, 在发送的画面绘制 GDI 文字。函数采用以 RGB() 宏形式的 COLORREF, 或者 256 色的 8 位颜色索引。注意函数操作时, 目标画面不能够被锁定, 因为使用 GDI 执行文字变换时, 需要短暂地锁定它。成功则返回 TRUE。

例子:

```
// draw text with color RGB(100,100,0);
Draw_text_GDI("This is a test", 100,50,
              RGB(100,100,0),lpddsprimary);
```

数学和误差函数

数学库函数目前几乎不存在, 但是一旦你到了本书的数学部分, 它马上就改变。我将向你的大脑充入大量有趣的数学信息和函数。到那时, 饱尝简单的甜头吧.....

函数原型:

```
Int Fast_Distance_2D(int x, int y);
```

目的:

Fast_Distance()用快速方法计算从(0, 0)到(x, y)的距离。返回一个有 3.5%误差的取整

距离。

例子:

```
int x1=100, y1=200; // object one
int x2=400, y2=150; // object two

// computer the distance between object one and two
int dist = Fast_Distance_2D(x1-x2, y1-y2);
```

函数原型:

```
float Fast_Distance_3D(float x, float y, float z);
```

目的:

Fast_Distance_3D()采用快速方式计算从(0, 0, 0)到(x, y, z)的距离。函数返回一个有11%误差的距离。

例子:

```
// compute the distance from(0,0,0) to (100,200,300)
float dist = Fast_Distance_3D(100,200,300);
```

函数原型:

```
int Find_Bounding_Box_Poly2D(
    POLYGON2D_PTR poly, // the polygon
    Float &min_x, float &max_x, // bounding box
    Float &min_y, float &max_y);
```

目的:

Find_Bounding_Box_Poly2D()计算包含通过参数 poly 给定的多边形的最小矩形。成功返回 TRUE。注意, 函数采用带参参数。

例子:

```
POLYGON2D poly; // assume this is initiatlized
Int min_x, max_x, min_y, min_y; // hold result

// find bounding box
Find_Bounding_Box_Poly2D(&poly,min_x,max_x, min_y, min_y);
```

函数原型:

```
Int open_Error_File(char *filename);
```

目的:

Open_Error_File()函数打开一个接受函数 Write_Error()发送的误差信息的文件。成功返回 TRUE。

例子:

```
//open a general error log
Open_Error_File("errors.log");
```

函数原型:

```
Int Close_Error_File(void);
```

目的:

Close_Error_File()关闭前面打开的错误信息文件。一般情况下，它关闭这个流。如果你调用它，而误差信息文件没有打开，则什么都不做。成功返回 TRUE。

例子:

```
// close the error system, note no parameter needed
Close_Error_File();
```

函数原型:

```
Int Write_Error (char *string,...); // error formatting string
```

目的:

Write_Error()函数在前面打开的误差信息文件中写入误差信息。如果没有打开文件，函数返回 FALSE 并不作任何改动。注意，函数使用了可变个数的参数，可以像使用 printf()那样使用它。成功返回 TRUE。

例子:

```
// write out some stuff
Write_Error("\nSystem Starting...");
Write_Error("x-vel = %d", y-vel = %d", xvel, yvel);
```

图形函数

下面的函数组成了 BITMAP_IMAGE 和 BITMAP_FILE 操作线。有函数调用 8、16、24、32 位位图，有的从它们中提取图像创建 BITMAP_IMAGE 对象（非 DirectDraw 画面）。另外，还具有绘制这些图像的功能，但是不支持剪切。因此，如果你需要剪切，或者想建立在最后部分要讨论的 BOB 对象，你自己可以对源进行更改。

函数原型:

```
Int Load_Bitmap_File(BITMAP_file_ptr BITMAP, // BITMAP FILE
                     Char *filename); // disk .BMP file to load
```


目的:

`Load_Bitmap_File()`将磁盘中的一张位图(.BMP)文件,调入要发送的 `BITMAP_FILE` 结构结构中。函数装载 8、16、24 位位图以及 8 位.BMP 文件的调色板信息。成功返回 `TRUE`。

例子:

```
// let's load "andre.bmp" off disk
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "andre.bmp");
```

函数原型:

```
Int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap);
// bitmap to close and unload
```

目的:

`Unload_Bitmap_File()`函数释放装载的 `BITMAP_FILE` 的图像缓冲占用的内存。当你已经拷贝了图像的位且/或采用特殊位图工作时,调用该函数。你可以重新使用该结构,但是,必须首先释放内存。成功返回 `TRUE`。

例子:

```
// close the file we just opened
Unload_Bitmap_File(&bitmap_file);
```

函数原型:

```
Int Create_Bitmap(BITMAP_IMAGE_PTR image, // bitmap image
Int x, int y, // starting position
Int width, int height); // size
```

目的:

`Create_Bitmap()`用给定的尺寸和位置创建 8 位系统内存图像。图像开始是黑色的,存储在 `BITMAP_IMAGE` 中。图像不是 `DirectDraw` 画面,所以不能使用剪切。成功返回 `TRUE`。

注 意

`BITMAP_IMAGE` 和 `BITMAP_FILE` 之间有很大差异。`Bitmap_File` 是磁盘上的.bmp 文件,而 `BITMAP_IMAGE` 是系统内存对象,可以被移动和绘制。

例子:

```
// let's create a 64X64 bitmap image at (0,0)
BITMAP_IMAGE ship;

Create_Bitmap(&ship, 0,0,64,64);
```

函数原型:

```
int Destroy_Bitmap(BITMAP_IMAGE_PTR image); // bitmap image to destroy
```

目的:

Destory_Bitmap()用来释放由于创建 **BITMAP_IMAGE** 对象而分配的内存。你应该在使用完毕该对象之后调用这个函数——通常在游戏结束时，或者这个对象在血腥的战斗中被消灭后。成功返回 **TRUE**。

例子:

```
// destroy the previously created BITMAP_IMAGE
Destroy_Bitmap(&ship);
```

函数原型:

```
Int Load_Image_Bitmap(
    BITMAP_IMAGE_PTR image, // bitmap to store image in
    BITMAP_FILE_PTR bitmap, // bitmap file object to load from
    Int cx, int cy, // coordinates where to scan (cell or abs)
    Int mode); // image scan mode: cell based or absolute

#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS 1
```

目的:

Load_Image_Bitmap()从前面装载的 **BITMAP_FILE** 对象中扫描一个图像存储在 **BITMAP_IMAGE** 存储区，这是让对象或者图像位进入一个 **BITMAP_IMAGE** 中的方法。现在要使用它，首先要装载一个 **BITMAP_FILE** 并创建一个 **BITMAP_IMAGE**。然后，调用它从存储在 **BITMAP_FILE** 中的位图数据中获得一个同样大小的扫描的图像。函数工作有两种方式：单元格模式或绝对坐标模式。

- 单元格模式，**BITMAP_EXTRACT_MODE_CELL**，图像的扫描建立在下面的假设之上：假设所有的图像都以一个给定尺寸 $m \times n$ 的模板，在单元之间有一个像素的边界，存储在 **.BMP** 文件中。单元通常是 8×8 、 16×16 、 32×32 、 64×64 等大小。看 CD 中的 **TEMPLATE*.BMP**，它包含几个模板，单元数从上至下，从左至右，从(0, 0)开始排列。
- 第二种模式是绝对坐标模式，**BITMAP_EXTRACT_MODE_ABS**。这种模式下，图像在参数 **cx**、**cy** 传送的精确坐标下被扫描。对于同一个 **.BMP**，如果想以不同尺寸的图像来装载你的作品，这是一个好方法。因此，你不能将它们模板化。

例子:

```
// assume the source bitmap .BMP file is 640×480 and
```

```
// has a 8x8 matrix of cells that are each 32x32
// then to load the 3rd cell to the right on the 2nd
// row(cell 2,1), you would do this

// load in the .BMP file into memory
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file, "images.bmp");

// initialize the bitmap
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0,0, 32,32);

// now scan out the data
Load_Image_Bitmap(&ship, &bitmap_file, 2,1,
                  BITMAP_EXTRACT_MODE_CELL);
```

要准确装载相同的图形，假设该图形仍然在模板中，使用绝对坐标模式，必须指出其坐标。请记住，图形每条边有厚度为一个像素的分界。

```
Load_Image_Bitmap(&ship, *bitmap_file,
                  2*(32+1)+1,1*(32+1)+1,
                  BITMAP_EXTRACT_MODE_ABS);
```

函数原型:

```
Int Draw_Bitmap(BITMAP_IMAGE_PTR source_bitmap, // bitmap to draw
                UCHAR *dest_buffer, // video buffer
                Int lpitch, // memory pitch
                Int transparent); // transparency?
```

目的:

Draw_Bitmap()采用透明或者不透明的方式，在目标内存画面绘制传送的位图。如果 transparency 是 1，透明被允许，具有颜色索引 0 的任何像素都不被拷贝。成功返回 TRUE。

例子:

```
// draw our little ship on the back buffer
Draw_Bitmap(&ship, back_buffer, back_lpitch, 1);
```

函数原型:

```
Int Flip_Bitmap(UCHAR *image, // image bits to vertically flip
                Int bytes_per_line, // bytes per line
                Int height); // total rows or height
```

目的:

Flip_Bitmap()通常在内部使用，用来将颠倒的.bmp 文件翻转过来，但你也可以用它翻转一个图像。函数在内存进行翻转操作，实际上是一行一行把图颠倒过来，所以你最初发

送的数据被颠倒，这一点请注意。函数成功则返回 TRUE。

例子:

```
// for fun flip the image bits of our little ship
Flip_Bitmap(ship->buffer, ship->width, ship_height);
```

调色板函数

下面的函数构成 256 色调色板接口。这些函数只是在你的显示器设为 256 色，即 8 位颜色时才有用。

函数原型:

```
int Set_Palette_Entry(
    int color_index, // color index to change
    LPPALETTEENTRY color); // the color
```

目的:

Set_Palette_Entry()被用来改变在调色板中的某一颜色。只需发送 0~255 颜色索引和存放颜色的 PALETTEENTRY 指针，就可以在下一帧刷新画面的颜色。另外，该函数也可用于刷新阴影调色板。注意，函数运行较慢。如果想调整整个调色板，请使用 Set_Palette()。函数成功则返回 TRUE，失败返回 FALSE。

例子:

```
// set color 0 to black
PALETTEENTRY black = {0,0,0,PC_NOCOLLAPSE};
Set_Palette_Entry(0,&black);
```

函数原型:

```
int Get_Palette_Entry(
    int color_index, // color index to retrieve
    LPPALETTEENTRY color); // storage for color
```

目的:

Get_Palette_Entry()从当前调色板获得一个调色板接口。但是，该函数非常快，因为它基于 RAM 的阴影调色板。因此，只要你喜欢就可以调用它，因为它不打扰硬件的运行。但是，如果你采用 Set_Palette_Entry()或者 Set_Palette()函数修改了系统调色板，这个阴影调色板将不能够被刷新，而且所获得的数据也可能无效。成功则函数返回 TRUE，失败返回 FALSE。

例子:

```
// let's get palette entry 100
```

```

PALETTEENTRY color;
Get_Palette_Entry(100,&color);
Int Save_Palette_Entry(100,&color);

```

函数原型:

```

Int Save_Palette_To_File(
    Char *filename, // filename to save at
    LPPALETTEENTRY palette); // palette to save

```

目的:

Save_Palette_To_File()函数将传递的调色板数据保存到磁盘上的一个 ASCII 文件中, 以利于以后的检索或处理。如果你能产生了一个运行中的调色板, 并想保存它, 该函数就非常方便。但是, 函数假设调色板的指针指向一个 256 入口的调色板, 所以要当心。成功则函数返回 TRUE, 失败返回 FALSE。

例子:

```

PALETTEENTRY my_palette[256]; // assume this is built

// save the palette we made
// note file name can be anything, but I like *.pal
Save_Palette_To_file("/palettes/custom1.pal",my_palette);

```

函数原型:

```

Int Load_Palette_From_File(
    Char *filename, // file to load from
    LPPALETTEENTRY palette); // storage for palette

```

Load_Palette_From_File()用来将前面通过 Save_Palette_To_File()存储在磁盘上的 256 色调色板取出来。你只需要发送一个文件名和一个用来存储 256 调色板的存储结构, 从磁盘上调出的调色板就存放在数据结构之中了。但是函数不能把调色板装入硬件调色板, 你自己还需要通过 Set_Palette()完成这一步。成功则函数返回 TRUE, 失败返回 FALSE。

例子:

```

// load the previously saved palette
PALETTEENTRY disk_palette[256];

Load_Palette_From_Disk("/palettes/custom1.pal", &disk_palette);

```

函数原型:

```

Int Set_Palette(LPPALETTEENTRY set_palette);
// palette to load into hardware

```

目的:

Set_Palette()将发送来的调色板数据装入硬件,也可调整阴影调色板。成功则函数返回 TRUE,失败返回 FALSE。

例子:

```
// lets load the palette into the hardware
Set_Palette(disk_palette);
```

函数原型:

```
Int Save_Palette(LPPALETTEENTRY sav_palette); // storage for palette
```

目的:

Save_Palette()将硬件调色板扫描到 sav_palette 中,使你能够操作它或者保存到磁盘。sav_palette 必须有足够存储所有 256 入口的空间。

例子:

```
//retrieve the current DirectDraw hardware palette
PALETTEENTRY hardware_palette[256];
Save_Palette(hardware_palette);
```

函数原型:

```
Int Rotate_Colors(int start_index, // starting index 0..255
                  Int end_index); // ending index 0..255
```

目的:

Rotate_Colors()以循环方式旋转一簇颜色。它直接操纵硬件调色板。成功则函数返回 TRUE,失败返回 FALSE。

例子:

```
// rotate the entire palette
Rotate_Colors(0,255);
```

函数原型:

```
Int Blink_Colors(int command, // blinker engine command
                 BLINKER_PTR new_light, // blinker data
                 Int id); // id of blinker
```

目的:

Blink_Colors()用于创建一个异步调色板动画。函数在此解释起来太长,请参考第七章“高级 DirectDraw 和位图图形”寻求更详尽的描述。

例子:

无。

实用函数

下面是我用得比较多的实用函数，所以认为，你也会用得比较多。

函数原型:

```
DWORD Get_Clock(void);
```

目的:

Get_Clock()以毫秒为单位，返回自从 Windows 启动后的时间。

例子:

```
// get the current tick count
DWORD start_time = Get_Clock();
```

函数原型:

```
DWORD Start_Clock(void);
```

目的:

Start_Clock()调用 Get_Clock()并将时间存到一个全局变量中。然后可以调用 Wait_Colck(), 将从调用 Start_Clock()开始等待一定时间（以毫秒计）。函数返回调用开始时的时钟值。

例子:

```
// start the clock and set the global
Start_Clock ();
```

函数原型:

```
DWORD Wait_Clock(DWORD count);
```

目的:

Wait_Clock()等待从调用 Start_Clock()后发送来的以毫秒计的时间。函数返回调用时的时钟值。但是，要等待到剩余时间被消耗完为止才返回。

例子:

```
// wait 3 milliseconds
Start_Clock();

// code...
```

```
Wait_Clock(30);
```

函数原型:

```
int Collision_Test(int x1, int y1, // upper lhs of obj1
                  int w1, int h1, // width, height of obj1
                  int x2, int y2, // upper lhs of obj2
                  int w2, int h2, // width, height of obj2)
```

目的:

Collision_Test()主要执行发送来的两个矩形重叠部分矩形的测试。发送来的矩形可以代表你喜欢的任何东西。你必须给定它们的左上角坐标和宽度及高度。如果有重叠返回 TRUE, 否则返回 FALSE。

例子:

```
// do these two BITMAP_IMAGE's overlap?
If (Collision_Test(ship1->x, ship1->y, ship1->width, ship1->height,
Ship2->x, ship2->y, ship1->width, ship1->height))
{ // hit

} // end if
```

函数原型:

```
int Color_Scan(int x1, int y1, // upper lhs of rect
               int x2, int y2, // upper lhs of rect
               UCHAR scan_start, // starting scan color
               UCHAR scan_end, // ending scan color
               UCHAR *scan_buffer, // memory to scan
               int scan_lpitch); // linear memory pitch
```

目的:

Color_Scan()是另外一种碰撞测试算法, 在一些连续区域扫描一个矩形以寻找某 8 位颜色值或者值顺序。你可以用它检测在某个地方是否存在一个颜色索引。当然, 它仅仅适用于 8 位模式, 但是, 函数不难扩展为 16 位或更高的色彩模式。如果颜色找到返回 TRUE。

例子:

```
// scan for colors in range from 122-124 inclusive
Color_Scan(10, 10, 50, 50, 122,124,back_buffer, back_lpitch);
```


BOB (变换对象) 引擎

虽然采用位编程可以得到你想要的 `BITMAP_IMAGE` 类型, 但是, 这样有一个严重的缺点——不能够使用 `DirectDraw` 画面, 也就不支持加速功能。因此, 我创建了一个称为 BOB (变换对象) 的类似精灵的类型。对于你这些已经在游戏程序的山洞中存在的小东西, 精灵实际上是你移动而不改变背景的对象。也不总是这样, 所以我称我的动画小对象为 BOB 而不是精灵。

迄今为止, 你在本书中没有见到任何 BOB 引擎代码, 但是, 却已经知道完成它的所有步骤。我没有时间将所有的源代码列出, 它同其他的東西一起, 在文件 `T3DLIB1.CPP` 中。我在这里想给你介绍的是组成引擎的每个函数。你可以免费使用这些代码, 打印或者销毁它等等。在你转到剩余的 `DirectX` 非图形组件之前, 我只是想给你一个使用 `DirectDraw` 画面和所有加速的例子。

让我们简单地谈谈 BOB 是什么。首先, 在后面数据类型和结构部分, 看看 BOB 的数据结构, 然后再回到这里……准备好了吗?

BOB 一般是由一个或一些 `DirectDraw` 画面 (最大到 64) 代表的图形对象。你可以移动 BOB、绘制 BOB 或者做动画 BOB。BOB 是由当前 `DirectDraw` 剪切板剪切的, 所以, 它们剪切的同时也被加速——这是一件好事。图 8.55 给出了 BOB 同它的动画画面的关系。

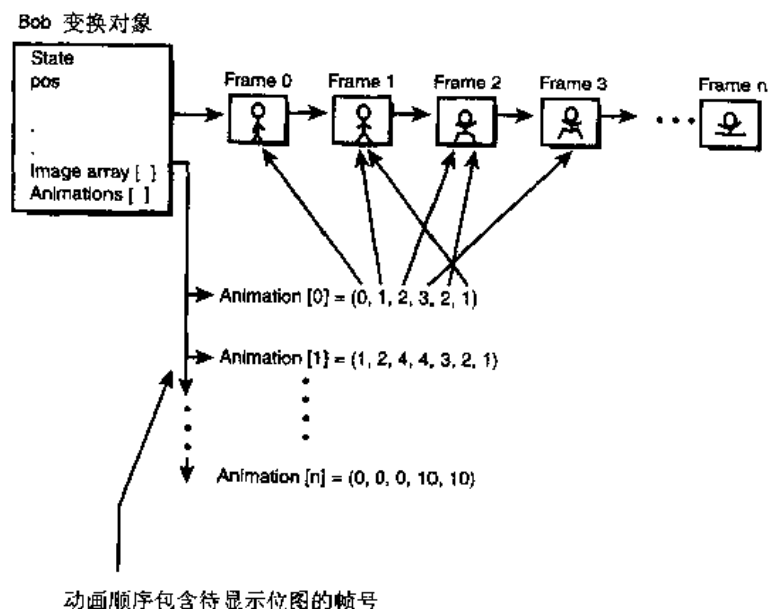


图 8.55 BOB (变换对象) 动画系统

BOB 引擎也支持动画序列, 所以你可以以一套画面、一个动画序列的方式调用它, 动

画序列由画面组成。这是非常酷的特征。同样，BOB 函数成功返回 TUN，否则返回 FALSE。让我们来看看它们。

函数原型:

```
int Create_BOB(int BOB_PTR bob, // ptr to bob to create
    int x, int y, // initial position of bob
    int width, int height, // size of bob
    int num_frames, // total number of frames for bob
    int attr, // attributes of bob
    int mem_flags); // surface memory flags, 0 is VRAM
```

目的:

Create_BOB() 创建一个 BOB 对象，并设置它。函数设置所有的内部变量另外还为每一个画面创建单独的 DirectDraw 画面。多数参数是自解释的：惟一需要一点解释的是特征变量 attr。表 8.4 较好地描述了每一个属性，你可以将它们一起逻辑“或”之后发送到字段。

表 8.4 有效的 BOB 属性

值	描 述
BOB_ATTR_SINGLE_FRAME	使用单画面创建一个 BOB
BOB_ATTR_MULTI_FRAME	使用多画面创建一个 BOB，但是 BOB 动画以线性次序依次通过画面 0...n
BOB_ATTR_MULTI_ANIM	创建一个多画面的支持动画次序的 BOB
BOB_ATTR_ANIM_ONE_SHOT	设定了该标志，将只运行一次动画，然后停止。此时内部变量 anim_state 设定。要再次运行动画，则重新设置该变量
BOB_ATTR_BOUNCE	该标志通知 BOB 像一个球一样在屏幕边界上弹跳，只有在使用 Move_BOB() 时，该标志才工作
BOB_ATTR_WRAPAROUND	该标志通知 BOB 在移动时环绕屏幕的另一边。只有在使用 Move_BOB() 时，该标志才工作

例子:

这里是创建 BOB 的一些例子，首先是位于(50, 100)尺寸为 96×64 的单个画面的 BOB。

```
BOB car; // a car bob

// create the bob
if (!Create_BOB(&car, 50, 100,
    96, 64, 1, BOB_ATTR_SINGLE_FRAME, 0))
{ /* error */ }
```

下面是8画面的，大小为32×32的多画面BOB。

```
BOB ship; // a space ship bob

// create the bob
if (!Create_BOB(&ship, 0, 0,
                32, 32, 8, BOB_ATTR_MULTI_FRAME, 0))
{ /* error */ }
```

最后是支持动画序列的多画面BOB。

```
BOB greeny; // allitle green man bob

// create the bob
if (!Create_BOB(&greeny, 0, 0,
                32, 32, 8, BOB_ATTR_MULTI_ANIM, 0))
{ /* error */ }
```

函数原型:

```
int Drstroy_BOB(BOB_PTR bob); // ptr to bob to destroy
```

目的:

Destory_BOB()函数销毁前面所创建的BOB。在你使用BOB后，想把它所占用的内存还给Windows时，调用此函数。

例子:

```
// destroy the BOB above, you would do this
Destroy_BOB(&greeny);
```

函数原型:

```
int Draw_BOB(BOB_PTR bob, // ptr of bob to draw
             LPDIRECTDRAWSURFACE dest); // dest surface to draw on
```

目的:

Draw_BOB()是一个非常强大的函数。它在你给它的DirectDraw画面绘制发送来的BOB。BOB被绘制在它的动画参数定义的当前位置和当前框架中。

警告



要使用该函数，目标画面不能被锁定。

例子:

```
// this is how you would position a multiframe BOB at
// (50, 50) and draw the first frame of it on the back
// surface
BOB ship; // a space ship bob

// create the bob
if (!Create_BOB(&ship, 0, 0,
32, 32, 8, BOB_ATTR_MULTI_FRAME, 0))

// load the bob images in ... well get to this in a bit
// set the position and frame of bob
ship.x = 50;
ship.y = 50;
ship.curr_frame = 0; // this contains the frame to draw

// draw bob
Draw_BOB(&ship, lpddsback);
```

函数原型:

```
int Draw_Scaled_BOB(BOB_PTR bob, // ptr of bob to draw
int swidth, int sheight, // new width and height of bob
LPDIRECTDRAWSURFACE dest); // dest surface to draw on
```

目的:

Draw_Scaled_BOB()同 **Draw_BOB()**一样工作, 你还可以给要绘制的 BOB 发送一个新的宽度和高度。如果你有加速器, 这非常酷。它是一个使 BOB 看起来像三维对象的一个很好的方法。

例子:

```
// an example of drawing the ship 128X128 even though
// it was created as only 32X32 pixels
Draw_Scaled_BOB(&ship, 128, 128, lpddsback);
```

函数原型:

```
int Load_Frame_BOB(
BOB_PTR bob, // ptr of bob to load frame into
BITMAP_FILE_PTR bitmap, // ptr of bitmap file to scan data
int frame, // frame number to place image into 0, 1, 2...
int cx, int cy, // cell pos of abs to scan from
int mode); // scan mode, same as Load_Frame_Bitmap()
```

目的:

Load_Frame_BOB()同 **Load_Frame_Bitmap()**函数一样工作, 详细情况参考

Load_Frame_Bitmap()函数。惟一增加的控制参数 frame 是要装载的画面。如果你创建的 BOB 有四个画面，你将一个个装载它们。

例子:

```
// here's an example of loading 4 frames into a BOB from a
// bitmap file in cell mode

BOB ship; // the bob
// loads frames 0, 1, 2, 3 from cell position (0, 0), (1, 0)
// (2, 0), (3, 0)
// from bitmap8bit bitmap file, assume it has been loaded

for (int index=0; index<4; index++)
Load_Frame_BOB(&ship, &bitmap8bit,
index, index, 0
BITMAP_EXTRACT_MODE_CELL);
```

函数原型:

```
int Load_Animation)BOB(
BOB_PTR bob, // bob to load animation into
int anim_index, // which animation to load 0..15
int num_frames, // number of frames of animation
int *sequence); // ptr to array holding sequence
```

目的:

Load_Animation()需要解释一下。此函数将 16 数组的内容之一装载到含有动画序列的 BOB 中。每个序列包含一个索引的数组或者要顺序显示的画面数目。

例子:

你可能有一个具有 8 画面的 BOB, 0, 1, 2...7, 但是, 你可能有一个像下面这样定义的四帧动画。

```
int anim_walk[] = {0, 1, 2, 1, 0};
int anim_file[] = {5, 6, 0};
int anim_die[] = {3, 4};
int anim_sleep[] = {0, 0, 7, 0, 0};
```

然后, 要装载动画到 BOB, 需要:

```
// create a multi animation bob
// create the bob
if (!Create_BOB(&alien, 0, 0, 32, 32, 8, BOB_PTR_MULTI_ANIM,0))
{ /* error */}

// load the bob frame in...
```

```
// load walk into animation 0
Load_Animation_BOB(&alien, 0, 5, anim_walk);

// load fire into animation 1
Load_Animation_BOB(&alien, 1, 3, anim_fire);

// load die into animation 2
Load_Animation_BOB(&alien, 2, 2, anim_die);

// load sleep into animation 3
Load_Animation_BOB(&alien, 3, 5, anim_sleep);
```

在装载动画之后，你可以激活动画，使用函数播放它们，一会儿便可看到放映的动画。

函数原型:

```
int Set_Pos_BOB(BOB_PTR bob,    // ptr to bob to set position
                int x, int y); // new position of bob
```

目的:

Set_Pos_BOB()是一个设置 BOB 位置的简单方法。它除了分配一个内部(x, y)变量之外没有其他作用，但是这一功能的确有用。

例子:

```
// set the position of the alien BOB above
Set_Pos_BOB(&alien, player_x, player_y);
```

函数原型:

```
int Set_Vel_BOB(BOB_PTR bob,    // ptr to bob to set velocity
                int xv, int yv); // new x, y velocity
```

目的:

每一个 BOB 具有一个内部速度，在(xv, yv)中，Set_Vel_BOB()只是用函数发送来的新的值修改它。在 BOB 中的速度值没有任何作用，除非你用 Move_BOB()函数移动它。但是，即使你没有移动，也可以用(xv, yv)追踪 BOB 的速度。

例子:

```
// make the BOB move in a straight horizontal line
Set_Vel_BOB(&alien, 10, 0);
```

函数原型:

```
int Set_Anim_Speed_BOB(BOB_PTR bob,    // ptr to bob
                       int speed); // speed of animation
```

目的:

`Set_Anim_Speed()`为一个 BOB 的 `anim_count_max` 设置一个内部动画速度。这个数越大, 动画越慢; 数字越小 (最小为 0), 动画越快。但是, 此函数值和你是否使用内部 BOB 动作函数 `Animate_BOB()` 有关。当然, 你必须首先创建一个具有多画面的 BOB。

例子:

```
// set the rate to change frames every 30 frames
Set_Anim_Speed_BOB(&alien, 30);
```

函数原型:

```
int Set_Animation_BOB(
    BOB_PTR bob,    // ptr to bob to set animation
    int anim_speed); // index of animation to set
```

目的:

`Set_Animation_BOB()` 设置将要被 BOB 演示的当前动画。在前面 `Load_Animation_BOB()` 中, 你创建了四个动画。

例子:

```
// make animation sequence number 2 active
Set_Animation_BOB(&alien, 2);
```

注 意

这也将 BOB 动画复位到序列中的第一个画面。

函数原型:

```
int Animate_BOB(BOB_PTR bob); // ptr to bob to animate
```

目的:

`Animate_BOB()` 是一个 BOB 动作。一般, 你应该每画面调用一次该函数以刷新 BOB 的动作。

例子:

```
// erase everying...
// move everying...
// animate everying
Animate_BOB(&alien);
```

函数原型:

```
int Move_BOB(BOB_PTR bob); // ptr to bob to move
```

目的:

Move_BOB()移动 BOB 一个 xv, yv 位置。然后,根据其属性,可能使 BOB 从墙上反弹回来,包裹起来,或者什么都不做。同 **Animate_BOB()**相类似,你要在主循环中在 **Animate_BOB()**之前(或之后)调用它一次。

例子:

```
// animate bob
Animate_BOB(&alien);

// move it
Move_BOB(&alien);
```

函数原型:

```
int Hide_BOB(BOB_PTR bob); // ptr to bob to hide
```

目的:

Hide_BOB()简单地为 BOB 设置不可见标志,于是 **DRAW_BOB()**将不显示它。

例子:

```
// hide the bob
Hide_BOB(&alien);
```

函数原型:

```
int Show_BOB(BOB_PTR bob); // ptr to bob to show
```

目的:

Show_BOB()为 BOB 设置一个可见标志,使它能够被绘制(同 **Hide_BOB()**相反)。下面有一个隐藏和显示 BOB 的例子,因为你正在显示一个 GDI 对象或者一些东西,并且不希望 BOB 堵塞它。

例子:

```
Hide_BOB(&alien);
// make calls to Draw_BOB and GDI etc
Show_BOB(&alien);
```


函数原型:

```
int Collision_BOBS(BOB_PTR bob1, // ptr to first bob
                  BOB_PTR bob2, // ptr to second bob
```

目的:

Collision_BOBS()监测两个 BOB 的矩形边界是否重叠。这在游戏中可用来检测玩家的 BOB 是否撞到 BOB 火箭什么的。

例子:

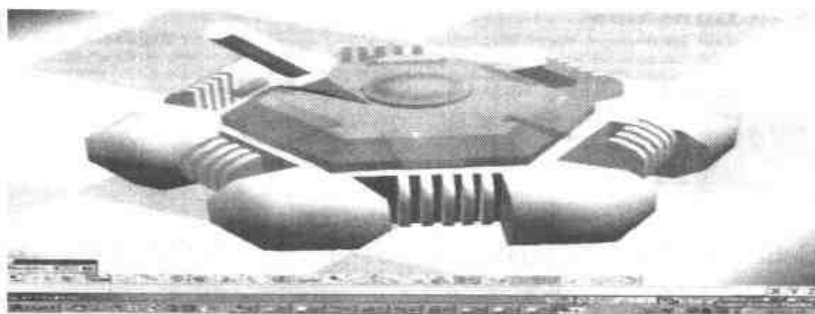
```
// check if a mission BOB hit a player BOB:
if (Collision_BOBS(&missile, &player))
{ /* make explosion sound */}
```

总结

本章花费了大量时间。覆盖内容太多，所以我想将不同主题的最重要的部分传授给你。但是，别灰心，到了 3D 的内容中，你将再次遇到所有的这些多边形内容，当这些都实现时，你将成为专家。

想想本章的主要议题：光栅化、剪切、直线绘制、矩阵、碰撞检测、计时、缩放、等角引擎等等。有时你不得不采用头朝下的方法，然后又得按照头朝上的方法回来。游戏编程就是这样，总之要头脑冷静。现在，你有了一个完整的图形和多边形函数库，用剩余章节中的演示程序，你可以做一些真正的“破坏”。让我们来享受其中的乐趣吧……

9



用 DirectInput 和力反馈进行输入

我记得有一次，我建立了一个 TTL 芯片输出操纵杆界面，使我在 Atari800 上编写的游戏能够每个 9 针操纵杆接口可以支持四个玩家。那使我累病了吗？无论如何，输入设备已经存在很久了，DirectX 应该支持它们。本章中，我们将看看 DirectInput 以及一些输入算法，并且，我将给出一个力反馈例子。这里是你将看到的：

- ➔ DirectInput 综述
- ➔ 键盘
- ➔ 鼠标
- ➔ 操纵杆
- ➔ 输入组合
- ➔ 力反馈
- ➔ 输入函数库

输入循环回顾

这是了解一个游戏的大致结构以及输入同事件循环关系的一个好机会。参见图 9.1。你看到擦除、移动、绘制、等待、重复等的一般游戏循环。对一个视频游戏来说，这就是所有的。当然，这有点过于简单。在 Win32/DirectX 世界中，我们添加了大量的设置，中断，Windows 事件处理代码，来确保一个 Windows/DirectX 程序正常运行。但是，所有这些经过仔细考虑，主要有擦除、移动、绘制、等待、重复。

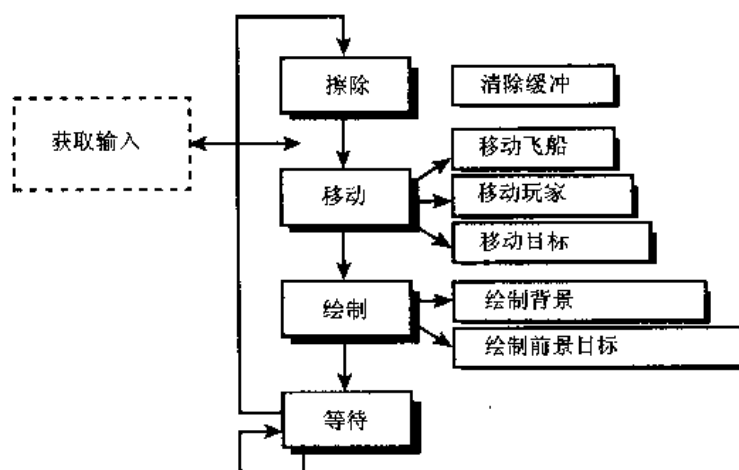


图 9.1 一般的输入循环

问题是：输入到哪里？问得好。你实际上可以把输入放到很多地方——在序列的开始、中间、结尾——但是，多数程序员喜欢把它恰好放在 move 部分之前。那样，玩家设置的最后输入状态就会在接下来的方框中执行。

图 9.2 给出了带有输入的更详细的游戏循环，并且所有的段更详细。记住，因为你正在使用用过的游戏控制台，所以你就已经同意了在 Game_Main() 函数中每个框架内做任何事。基本上，你的整个世界就在这一个函数调用中。

好了，现在我将刷新你的记忆，看输入应该在哪里被扫描和读取，让我们看看 DirectInput 是如何完成这件事的。

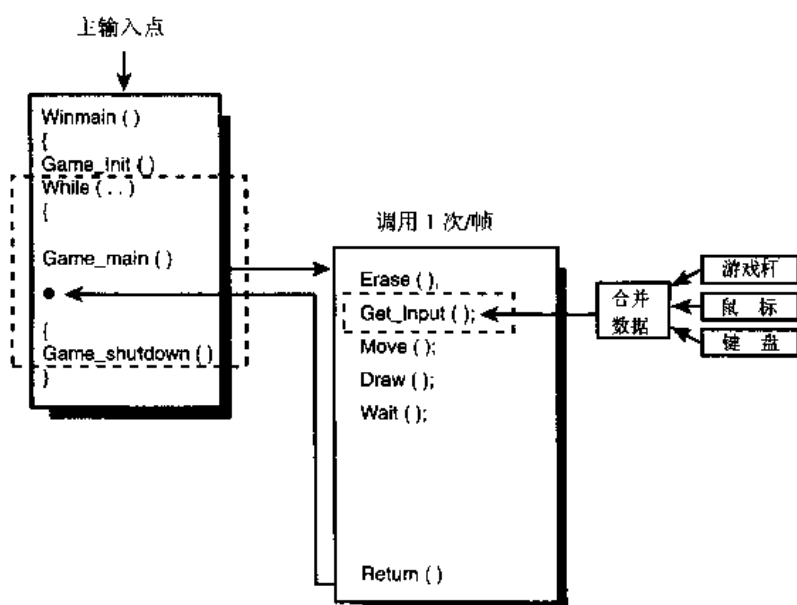


图 9.2 输入循环细节

DirectInput 序曲

DirectInput 同 DirectDraw 一样，是一个奇迹。没有 DirectInput，你肯定正在电话前，跟世界各地的输入设备制造商乞求驱动程序（DOS，Win16，Win32 等等，每样一个）呢，那将是一个糟糕的事情——相信我。DirectInput 将所有这些问题一扫而光。当然，由于它是由微软设计的，这也会产生一个新问题，但是它们至少都在一个公司里！

DirectInput 和 DirectDraw 一样。它是一个不依赖硬件的虚拟输入系统，它允许硬件制造商制造在统一接口下应用的传统的和非传统的输入设备。这对你很有好处，因为这使你不必拥有你的使用者可能拥有的每一个输入设备的驱动程序。你同 DirectInput 打交道，DirectInput 再把代码翻译成输入设备能够理解的代码，就像 DirectDraw 所做的那样。

如图 9.3 所示，它给出了 DirectInput 同硬件驱动程序以及物理输入设备之间的关系。

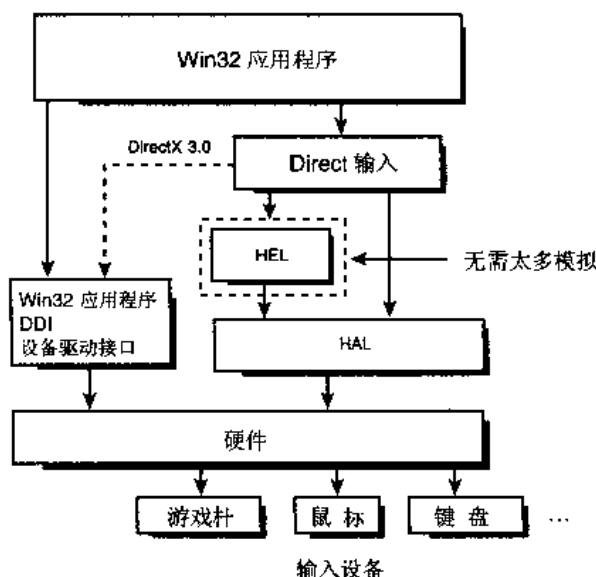


图 9.3 DirectInput 系统级概图

正如你看到的那样，你总是被 HAL(硬件抽象层)所隔离。需要用 HEL(硬件模拟层)进行模拟的东西不多，所以，它没有在 DirectDraw 中那样重要。无论何时，这是 DirectInput 的一个基本思想。我们来看看它支持什么。

各种输入设备

它确实是真的。只要有一个 DirectInput 驱动器，DirectInput 就可以同它交谈，所以你能。当然，需要制造商写一个驱动程序，但是这是他们的本职工作。记住这点，如你所

愿, DirectInput 支持下面的设备:

- 键盘
- 鼠标
- *操纵杆
- *踏板
- *游戏柄
- *舵轮
- *飞行操纵杆
- *头部追踪仪
- *6-DOF(自由度)空间球
- *电脑空间“性”服装(只要它们在 2002 年之前抢占了市场)

带星号的设备被 DirectInput 认为都是游戏杆。有许多类游戏杆子系统, DirectInput 统称它们为 devices。每一个这样的设备可以有一个或者多个输入对象, 可以是轴形的、转动的、临时的、摁的等等。知道了吗?例如, 一个游戏操纵杆可以有两个轴(X, Y)和两个临时开关共四个输入对象——就是这样。

DirectInput 并不关心设备是不是游戏操纵杆, 因为设备同样可以代表一个驾驶轮。但是, DirectInput 还是做了一点细分。不是鼠标或者键盘的任何东西都被认为是操纵杆, 不管你是手持、旋转、扭动或者脚踏。

DirectInput 在这些设备上的不同就迫使制造商(也就是驱动程序)提供一个 GUID 来对应它。这样, 每一个已经或者将要存在的设备都至少有一个特殊的名字。DirectInput 可以查询系统每个设备的所用的名字。但是, 一旦找到了这个设备, 它就只是—些输入对象。我将就此详细分析, 因为它迷惑了许多人。好吧, 现在让我们开始。

DirectInput 组件

DirectInput 包括一系列 COM 接口, 就像 DirectX 的任何子系统一样。参见图 9.4。你将看到主要的接口 IDirectInput 和惟一的其他设备接口 IDirectInputDevice(以及新版本的 IDirectInputDevice2)。

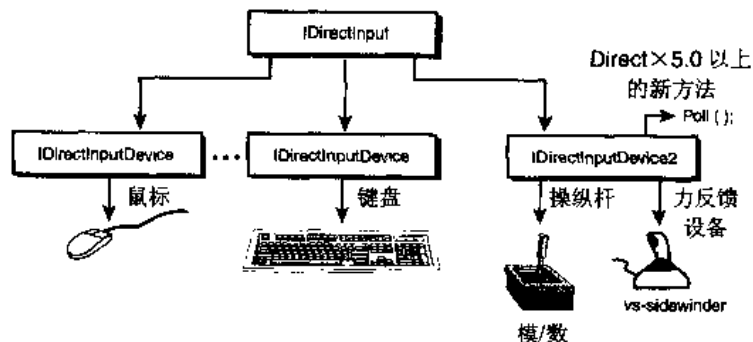


图 9.4 DirectInput 界面

下面来看看这些接口。

- **IDirectInput**——这是你启动 **DirectInput** 时必须创建的主 COM 对象。有一个软件包完成 **IDirectInputCreate()**，为你提供所有的 COM 材料。一旦创建了 **IDirectInput** 接口，你将通过查询它，设置 **DirectInput** 的属性，创建或者获得你想采用的任何输入设备。
- **IDirectInputDevice**——这个接口是从主 **IDirectInput** 接口创建的，它是你可以采用的任何设备的通信管道，不管它是一个鼠标、键盘、操纵杆或者是其他东西。它们都是 **IDirectInput** 设备。
- **IDirectInputDevice2**——它是 **IDirectInputDevice** 的改进版本，在 **DirectX 5.0** 之后创建。新的版本包括旧的 **IDirectInputDevice** 的所有功能，而且加上了对操纵杆和力反馈设备的支持。另外，它允许插拔查询注册设备，有些操纵杆需要查询注册。

在 **DirectX 5.0** 和 **IDirectInputDevice2** 之前，**DirectInput** 支持的操纵杆限定在 **WIN32** 驱动程序，也就是，根本不支持！如果你想用这个接口，从前面创建的 **IDirectInputDevice** 接口用 **QueryInterface()** 函数查询。（你还记得它吗？当我们到达这里时，我再演示给你。）

设置 **DirectInput** 的一般步骤

在运行 **DirectInput** 同一个或多个设备连接，最后从设备获取数据的时候，有几个步骤。首先是 **DirectInput** 的设置，接着是每一个输入设备的设置。第二部分对每一个设备几乎都是一样的，因此我们可将其归纳起来。

1. 通过调用 **DirectInputCreate()** 创建主 **DirectInput** 接口。这返回 **IDirectInput** 接口。

2. (可选) 查询设备 GUIDS。这步中，你将从 **DirectInput** 查询属于鼠标、键盘、操纵杆或其他通用设备(在前面没有枚举的)。这将伴随(准备放弃)一个回调函数和枚举量。一般查询 **DirectInput** 枚举某一类型/子类型的所有设备。**DirectInput** 通过一个复查过滤它们，然后你就可以建立一个 GUID 数据库。反感吧？幸运的是这只是操纵杆之类设备需要的事情，因为你可以使用一个通用的鼠标和键盘，对它们有常备的 GUID。等到了操纵杆时再讲述这一步。

3. 对你想在游戏中使用的每一个设备，创建时必须调用 **CreateDevice()** 传递一个 GUID。**CreateDevice()**，它是 **DirectInput** 的接口函数，所以你在进行此次调用之前，必须获得 **IDirectInput** 接口。如果你不知道你想创建的设备的 GUID，这一步就只有在第二步之后了。对于键盘和鼠标有两个固定的 GUID。

GUID_SysKeyboard——这是全局定义的总是作为主键盘设备的 GUID。

GUID_SysMouse——这是全局定义的总是作为主鼠标设备的 GUID。

提示



在前一段时间，有位聪明的读者发电子邮件问我如何检测和使用多个鼠标。我还没有准备好回答这个问题，但是，如果驱动程序支持多个鼠标，你应该在 **DirectInput** 下面使用它。这时，你应该查询备用鼠标的 GUID 来创建它。

最后,如果你想使用 `IDirectInputDevice2` 接口,现在就是查询它的时候。如果你想通过调用 `IDirectInputDevice::QueryInterface()`改善接口,你应该从 `IDirectInputDevice2::` 接口查询 `IDirectInputDevice2`。在你设置协作等级之前必须完成这个设置。

4. 一旦你创建了设备,你就必须设置协作等级。这由 `IDirectInputDevice::SetCooperativeLevel()`来完成。这里要注意 C++ 语法,它只是意味着 `SetCooperativeLevel()`是 `IDirectInputDevice` 的一个接口。协作等级同 `DirectDraw` 的相同,但是种类更少。但我们到了键盘的例子的时候,能够更仔细地看看它们。

5. 从 `IDirectInputDevice` 接口调用 `SetDataFormat()`函数,设置每一个设备的数据格式。这在实际应用中有点迷糊人,但是概念上还好理解。数据格式是你想为将要格式化的每个设备事件分配一个什么样的数据包。这就是 `DirectInput` 的好处,它将使你更灵活。谢天谢地,你可以使用一些已经定义好了的相当精彩的数据格式,所以,而不必自己来做这件事。

6. 用 `IDirectInputDevice::SetProperty()`设置你想要的任何设备的性能。这是对设备描述表敏感的,也就是说,一些设备有的性能另外一些设备就可能没有。所以,你必须知道你想设置什么。这次,你只是为操纵杆设备设置特性,但是,要清楚,在一个设备上认可的任何一个东西都要通过调用 `SetProperty()`实现。像以往一样,这个调用相当恐怖,当我们到了操纵杆例子的时候,我将讲述正确的步骤。

7. 通过调用 `IDirectInputDevice::Acquire()`获得每一个设备。这只是将你的设备同应用程序相连接并告诉 `DirectInput`,将来你要从这些设备中获取数据。

8. (可选)通过调用 `IDirectInputDevice::Poll()`轮询设备(注意,我在这个例子中使用 `IDirectInputDevice2` 接口来说明轮询只是和 `IDirectInput2` 同时生效)。有些设备需要被轮询而不是产生中断,并保持输入状态当前化。许多操纵杆在这一步出错,所以不管需要不需要,总是轮询它们是一个好主意。如果在不需要的情况下轮询了,将没有任何伤害,也不消耗什么(函数只是返回)。但是,正如我提到的那样,你只能够从接口的版本 2 轮询。

9. 调用 `IDirectInputDevice::GetDeviceState()`从每一个设备获取数据。每个不同设备返回的数据不同,但是调用是一样的。这个调用可以从设备获取数据并存在缓冲中以便你能够使用。

完毕!看起来很多,但事实上是为利用任意输入设备而不必担心它们的驱动程序付出的小代价。

数据采集模式

下面,我想简要地提醒你注意即时和缓冲数据模式。`DirectInput` 可以为你发送即时状态信息或者缓冲输入及以消息形式的时间标记。我使用缓冲输入的不多,如果你需要使用它,它也确实存在(如果感兴趣请参阅 `DirectX SDK`)。我们将使用数据获取的即时模式,也就是默认模式。

创建主 DirectInput 对象

现在，让我们看看如何创建主 DirectInput COM 对象 IDirectInput。然后我们将看看如何利用键盘、鼠标和操纵杆工作。

指向主 DirectInput 对象的接口指针定义在 DINPUT.H 中，如下所示：

```
LPDIRECTINPUT lpdi; // main directinput interface
```

要创建主 COM 接口，使用标准的函数 DirectInputCreate()，如下所示：

```
HRESULT WINAPI DirectInputCreate(
    HINSTANCE hInst, // the main instance of the app
    DWORD dwVersion, // the version of directinput you want
    LPDIRECTINPUT *lplpDirectInput, // ptr to storage
                                // for interface ptr
    LPUNKNOWN punkOuter); // COM stuff, always NULL
```

其中参数如下：

hInst 是你的应用程序的实例句柄。这是需要这个句柄的少数函数中的一个。它和你的程序开始时传递给 WinMain() 的实例句柄是同一个参数，所以，在这里把它存为全局变量并填充它。

dwVersion 是你想采用的 DirectInput 版本号常量。如果你假定你的一些程序将在 DirectX 3.0 下运行，这一点就要注意，但是只要把 DirectInput 的最新版本号送给 DIRECTINPUT_VERSION 就行了。

lplpDirectInput 是接口指针的地址，它接收 COM 接口给 DirectInput。

最后，**punkOuter** 是 COM 集合，没有用上，将它设为 NULL。

成功后 DirectInputCreate() 返回 DI_OK，不成功则返回其他东西。但是，像以往那样，我们将使用宏 SUCCESS() 和 FAILURE()，而不是检测 DI_OK，因为它在 DirectX 下检测问题时表现很好。不过，如果你愿意，使用 DI_OK 更安全。

这里是创建主 DirectInput 对象的例子。

```
#include "DINPUT.H" // need this and Dinput.LIB
// the rest of your includes, defines etc.
// globals...
LPDIRECTINPUT lpdi = NULL; // used to point to com interface

// create the main DirectInput object
if (FAILED(DirectInputCreate(main_instance,
    DIRECTINPUT_VERSION,
    &lpdi, NULL))
{ /* error */}
```


注 意

在应用程序中包含上 DINPUT.H 和 DINPUT.LIB 很重要, 否则, 编译器和连接器将不知道要干什么。如果你还没有读过我关于编译的介绍, 请在你的工程中加入 LIB 目录。仅在库函数搜索设置中设置搜索路径通常是不够的。

就是这样。如果函数成功, 此时你就有一个指向主 DirectInput 的指针, 然后你可以使用它创建设备。

对于所有 COM 对象, 在应用程序完成后, 要释放资源, 必须调用 Release() 递减 COM 对象的形参计数器。如下:

```
// the shutdown
lpdi->Release();
```

想更专业一点的话, 就这样:

```
// the shutdown
if (lpdi)
    lpdi->Release();
```

当然, 你需要在将创建的设备释放之后做这件事。记住总是要采用同创建相反的顺序调用 Release(), 就像解包裹一样。

101 键盘

因为在 DirectInput 中设置设备是触类旁通的。我将详细介绍键盘的设置, 然后大致介绍鼠标和操纵杆的设置。所以, 一定要仔细阅读这一部分, 因为它也适用于其他设备。

创建键盘设备

在使任何设备正常工作之前, 必须调用 IDirectInput::CreateDevice() 创建它。记住, 函数一般给你一个你需要的、随后能够使用的某一设备接口(第一个例子是键盘), 让我们来看看这个函数。

```
HRESULT CreateDevice(
    REFGUID rguid, // ptr to the
                  // IDirectInputDevice
                  // interface to receive ptr
    LPUNKNOWN pUnkOuter); // COM stuff, always NULL
```

够简单的吧? 第一个参数 rguid, 是你想创建设备的 GUID。你可以查询一个感兴趣的 GUID, 或者对大多数通用设备使用默认值。

GUID_SysKeyboard——键盘。

GUID_SysMouse——鼠标。

注意



危险！记住，它们在 DINPUT.H 中，它和 DINPUT 一起一定要被包含进去。另外，对于所有的 GUID，你应该在应用程序头部，在所有的其他 INCLUDE 之前，使用 #define INITGUID(但是只能是一次)，就像在你的应用程序中使用头文件 OBJBASE.H 一样。你可以在应用程序中采用 DXGUID.LIB，但是 OBJBASE.H 更好。任何时候，你都可以参考 CD 中演示程序一章中的内容，看什么被包含什么没有被包含。它只是琐碎的细节之一。

第二个参数是新的接口接收器，当然，最后一个是 NULL。函数成功则返回 DI_OK，否则返回其他东西。

好了，基于新发现的 Create_Device() 知识，来看看你能不能创建键盘设备。首先要做的一件事是需要一个保存将通过调用创建的接口指针的变量。所有设备的类型是 IDirectInputDevice 或者 IDirectInputDevice2，但是键盘也不特殊，所以你可以避开接口的 1.0 版本。

```
IDirectInputDevice lpdikey = NULL; // ptr to keyboard device
```

现在，让我们从主 COM 对象调用 CreateDevice() 创建这个设备。下面是所有的代码，包括主 COM 对象的创建和所有必须的包含文件。

```
// this needs to come first
#define INITGUID

// includes
#include <OBJBASE.H> // need this one for GUIDS
#include <DINPUT.H> // need this for directinput and
                  // DINPUT.LIB

// globals...
IDirectInput lpdi = NULL; // used to point to com interface
IDirectInputDevice lpdikey = NULL; // ptr to keyboard device

// create the main Directinput object
if (FAILED(DirectInputCreate(main_instance,
                             LPDIRECTINPUT_VERSION,
                             &lpdi, NULL)))
{ /* error */}
// now create the keyboard device
if (FAILED(lpdi->CreateDevice(GUID_SysKeyboard, &lpdikey, NULL)))
{ /* error */}

// do all the other stuff...
```

现在 lpdikey 指向一个键盘设备，你可以通过调用接口的方法设置协作等级、数据格式

等等。当然，如果用完该设备，应该调用 `Release()` 将它释放。但是，这个调用将放在你要释放的主 `DirectInput` 对象 `lpdi` 之前，所以在程序结束之前加入如下代码：

```
// release all devices
if (lpdikey)
    lpdikey->Release();

// ...more device releases, joystick, mouse etc.
// now release main COM object
if (lpdikey)
    lpdikey->Release();
```

设置键盘协作等级

一旦创建了你的设备(这里是键盘)，就需要设置它的协作等级，就像主 `DirectDraw` 对象一样。但是在 `DirectInput` 中，可选项不多。表 9.1 给出了可能的协作标识。

表 9.1 `DirectInput SetCooperativeLevel()` 的合作标志

值	描 述
<code>DISCL_BACKGROUND</code>	当应用程序在后台或前台时能够使用 <code>DirectInput</code> 设备
<code>DISCL_FOREGROUND</code>	应用程序要求前台访问。如果前台访问允许，相关窗口移到后台中时，该设备自动不获得
<code>DISCL_EXCLUSIVE</code>	一旦要求该设备，其他应用程序都不会访问它。但是其他应用程序依然能够要求非独占地访问它
<code>DISCL_NONEXCLUSIVE</code>	应用程序请求非独占访问。访问该设备将不会干涉其他访问相同设备的应用程序

这些使我头疼。好像是：“后台，前台，独占，非独占——让我头疼”。但是，多见几次，就会很清楚不同标志的含义了。一般地，如果用 `DISCL_BACKGROUND`，你的应用程序将不管是激活或者是最小化都接收输入。如果设置 `DISCL_FOREGROUND`，则只有在你的应用程序在最上面时才发送给它。

独占/非独占设置你的程序是否总是控制设备，其他程序是否能够使用。例如，键盘和鼠标是最简单的独占设备，当你的应用程序捕捉到它们时，其他程序只有在焦点状态才能够使用它们。这制造了一些自相矛盾。

首先，你只能够使键盘工作在非独占模式下，因为 `Windows` 本身总是要能够获得 `Alt` 键组合。其次，如果想的话，可以使鼠标获得独占模式，但是，这样你将在你的应用程序中丢失鼠标消息(这可能是你想要的)并且光标消失。就像你所关心的，因为你可能想自己绘制一个。最后，多数力反馈操纵杆(通用操纵杆)应该工作在独占模式下。但是，你可以设置正常的操纵杆为非独占模式。

所以，这些情况下我们应将标志设为 `DISCL_BACKGROUND` 或 `DISCL_NONEXCLUSIVE`。需要你设成独占模式的惟一条件是具有力反馈设备。当然，这个设置可能会使你在其他应用程序被激活并要利用独占模式时丢失设备。那样，你就不得不重新获取设备，不过，这方面内容我们一会儿再谈。

现在，用 `IDIRECTINPUTDEVICE::SetCooperativeLevel()` 来设置协作等级。如下所示：

```
HRESULT SetCooperativeLevel(HWND hwnd, // the window handle
DWORD dwFlags); // cooperation flags
```

下面是调用需要为键盘设置协作等级的调用(所有的设备都采用统一的办法)：

```
if (FAILED(lpdikey->SetCooperativeLevel(main_window_handle,
DISCL_BACKGROUND | DISCL_NONEXCLUSIVE)))
{ /* error */}
```

不起作用的惟一可能是另外一个应用程序具有独占/前台模式且正在使用。此时你只有等待，或者关闭这个正在占据输入设备的程序。

设置键盘的数据格式

使键盘准备好输入数据的下一步是设置数据格式。这通过调用 `SetDataFormat()` 来完成。如下所示：

```
HRESULT SetDataFormat(LPCDIDATAFORMAT lpdf); // ptr to data format structure
```

坏事……惟一的参数就是问题所在，这里是数据的结构：

```
// directinput dataformat
typedef struct
{
    DWORD dwSize; // size of this structure in bytes
    DWORD dwObjSize; // size of DIOBJECTDATAFORMAT in bytes
    DWORD dwFlags; // flags: either DIDF_ABSAXIS or
                  // DIDF_RELAXIS for absolute or
                  // relative reporting
    DWORD dwDataSize; // size of data packets
    DWORD dwNumObjs; // number of objects that defined in
                  // the following array of object
    LPDIOBJECTDATAFORMAT rgodf; // ptr to array of objects
} DIDATAFORMAT, *LPDIDATAFORMAT;
```

这是你要创建的真正复杂的数据结构，对你的目的至关重要。它主要允许你在设备对象的等级下决定从输入设备得来的数据如何格式化。但是，幸运的是，`DirectInput` 具有几种常规的数据格式，它们可以在各种情况下工作，你只需使用其中的一种。表 9.2 列出了

这些格式。

表 9.2 DirectInput 常用的数据格式

值	描 述
c_dfDIKeyboard	通用键盘
C_dfDIMouse	通用鼠标
C_dfDIJoystick	通用游戏杆
C_dfDIJoystick2	通用力反馈

一旦你将数据格式设成其中的一种，DirectInput 将以一定格式发送每一个数据包。DirectInput 有一些预定义格式使其设置简化，见表 9.3。

表 9.3 使用通常数据格式发送数据时 DirectInput 的数据结构

值	描 述
DIMOUSESTATE	该数据结构含有鼠标消息
DIJOYSTATE	该数据结构含有一个标准的类游戏杆的设备消息
DIJOYSTATE2	该数据结构含有标准的力反馈设备消息

当我们到了鼠标和操纵杆的时候我将给你实际的数据结构。但是，你可能会问，这该死的键盘结构在哪儿呢？好，它太简单了，所以没有成为一种类型。它不过是一个 256 字节的数组。每一个代表一个键。这样使键盘看起来像一套 101 瞬间开关。

因此，使用默认的 DirectInput 数据格式和数据类型，同使用 WIN32 函数 GetAsyncKeyState() 非常类似。在任何时候，你所需要的一种类型都像下面这样：

```
typedef _DIKEYSTATE UCHAR[256];
```

技 巧



如果 DirectX 漏掉了一些功能，我想创建一个“DirectXish”版本，通常创建没有的数据结构或函数，但是，发明它从现在开始要六个月。

所以，记住这点，让我们设置小键盘的数据格式：

```
// set data format
if (FAILED(lpdikey->SetDataFormat(&c_dfDIKeyboard)))
{ /* error */ }
```

注意，我使用了&操作来获得全局变量 c_dfDIKeyboard 的地址，因为函数需要一个指向它的指针。

获取键盘

你快到达目的地了!快完成了!你已经创建了 `DirectInput` 主 COM 对象, 创建了设备, 设置了协作等级, 设置了数据格式。下一步就是从 `DirectInput` 获取设备。要做这件事, 需要调用 `IDIRECTINPUTDEVICE::Acquire()`, 无需参数。这里是例子:

```
// acquire the keyboard
if (FAILED(lpdikey->Acquired()))
{ /* error */}
```

现在都在这里了, 你准备从设备中获取输入吧。现在是该庆祝的时候了。我想我将拥有一个权杖

从键盘重载数据

从所有的设备重载数据都一样——加上或减去两个设备相关的细节。通常, 你需要做下面的工作:

1. (可选)像操纵杆一样从 `IDIRECTINPUTDEVICE2` 设备中轮询设备。
2. 通过调用 `IDIRECTINPUTDEVICE::GetDeviceState()` 或 `IDIRECTINPUTDEVICE2::GetDeviceState()` 从设备读取即时数据。

提示



记住, 任何可以通过 `IDIRECTINPUTDEVICE` 接口调用的方法也都可以通过 `IDIRECTINPUTDEVICE2` 调用。

`GetDeviceState()` 函数看起来如下所示:

```
HRESULT GetDeviceState(
    DWORD cbData,    // size of state datastructure
    LPVOID lpvData); // ptr to memory to receive data
```

第一个参数是重载数据的大小, 对于键盘是 256 字节, 对于鼠标是 `sizeof(DIMOUSESTATE)`, 对于游戏杆是 `sizeof(DIJOYSTATE)`, 等等。第二个参数是一个你将存储数据的指针。下面是从键盘读取数据:

```
// here's our little khelper typedef
typedef _DIKEYSTATE UCHAR[256];

_DIKEYSTATE keystate[256]; // this will hold the keyboard data
```

现在读取键盘:

```
if (FAILED(Jpdkey->GetDeviceState(sizeof(_DIKEYSTATE),
    (LPVOID) keystate)))
{ /* error */}
```

当然,你要在每个游戏循环之前在循环的顶部,在任何进程开始之前读取一次键盘。

一旦你有了数据,你就可以测试按键了,对吗?就像有一些常量对应函数 `GetAsynxKeyState()` 一样,也有一些常量对应键盘按钮,对应于它们在数组中的位置。它们都是以 `DIK_` 开头(我想是 `DirectInput Key` 的缩写),并且定义在 `DINPUT.H` 中。表 9.4 给出了它们中的一部分(请参考 `DirectX SDK` 查看整个列表)。

表 9.4 DirectInput 键盘状态常量

值	描 述
<code>DIK_ESCAPE</code>	ESC 键
<code>DIK_0~9</code>	主键盘 0~9
<code>DIK_MINUS</code>	减号键
<code>DIK_EQUALS</code>	等号键
<code>DIK_BACK</code>	空格键
<code>DIK_TAB</code>	Tab 键
<code>DIK_A~Z</code>	字母 A~Z
<code>DIK_LBRACKET</code>	左括号键
<code>DIK_RBRACKET</code>	右括号键
<code>DIK_RETURN</code>	主键盘上的 Return/Enter 键
<code>DIK_LCONTROL</code>	左 Control 键
<code>DIK_LSHIFT</code>	左 Shift 键
<code>DIK_RSHIFT</code>	右 Shift 键
<code>DIK_LMENU</code>	左 Alt 键
<code>DIK_SPACE</code>	空格键
<code>DIK_F1~15</code>	功能键 1~15
<code>DIK_NUMPAD0~9</code>	数字小键盘 0~9
<code>DIK_ADD</code>	数字小键盘上的 +
<code>DIK_NUMPADENTER</code>	数字小键盘上的 Enter
<code>DIK_RCONTROL</code>	右 Control 键
<code>DIK_RMENU</code>	右 Alt 键

续表

值	描 述
DIK_HOME	箭头键盘上的 Home 键
DIK_UP	箭头键盘上的上键
DIK_PRIOR	箭头键盘上的 PgUp 键
DIK_LEFT	箭头键盘上的左键
DIK_RIGHT	箭头键盘上的右键
DIK_END	箭头键盘上的 End 键
DIK_DOWN	箭头键盘上的下键
DIK_NEXT	箭头键盘上的 PgDn 键
DIK_INSERT	箭头键盘上的 Insert 键
DIK_DELETE	箭头键盘上的 Delete 键

注：黑体意味着一序列，如 DIK_0~9 表示有常量 DIK_0、DIK_1、DIK_2 等等。

要检测某键是否按下，必须检查该键的 8 位字节的 0x80 位;也就是最高位。例如，如果想检测 Esc 键是否按下，需要：

```
if (keystate[DIK_ESCAPE] & 0x80)
    { // it's pressederror */}
else
    { / * it's not */}
```

提 示



你不用&和位检测也许也能够通过，但是微软不保证在键没有被按下的情况下其他位是高还是低。进行位检测是一个安全的好主意。

“与”操作有点不好看，你可以用一个宏使它更好看一点，就这样：

```
#define DIKEYDOWN(data, n) (data[n] & 0x80)
```

然后只要：

```
if (DIKEYDOWN(keystate, DIK_ESCAPE))
    { // do it to it baby! */}
```

相当清楚了。当然，当你用完键盘的时候，你必须用函数 Unacquire()“反获取”，并且释放它(同时还有主 DirectInput 的 COM 对象)。像下面这样：


```

// unacquire keyboard
if (lpdikey)
    lpdikey->Unacquire();

// release all devices
if (lpdikey)
    lpdikey->Release();

// _more device unqcquire/release, joystick,mouse etc

// now releasr main COM object
if (lpdi)
    lpdi->Release();

```

这是我首次谈到函数 `Unacquire()`，但是，它同释放对象联系十分紧密，所以我觉得在这里使用合适。但是，如果你只是想“反获取”一个设备而不释放它，你就可以对这个设备使用 `Unacquire()`，并在以后需要的时候重新获取。有时你可能会在另外一个程序中利用你现在不用的设备，这时就可以这么做。

警告



当然，如果你想释放键盘而不释放操纵杆(或者其他组件)，那么在你准备完全去掉 `DirectInput` 之前就不要释放主 COM 对象。

使用键盘的例子参见 CD 上演示程序 `DEMO9_1.XPPIEXE`。图 9.5 给出了屏幕的一个动作图片。程序应用了我们设置键盘的所有技术，它允许你四处移动小人儿。编译这个程序，C++ 用户要记住包含 `DDRAW.LIB.DINPUT.LIB`。如果你查看程序开头部分，你会发现使用了 `T3DLIB1.H`。因此，在编译的过程中需要 `T3DLIB1.CPP`。在本章结束，我将给你一个完整的输入库函数(我已经完成了它，名字是 `T3DLIB2.CPP.H`，但是，我将在本章最后再给出)。

读数据过程中的问题：重获取

在 `DirectX` 使用中，我将不得不谈论一下可能存在的问题，因为总是有很多问题。它们不是缺陷，而是运行在协作 OS 环境（如 `Windows`）中的一种表现。使用 `DirectInput` 可能产生这类问题，这可能是由于设备引起的，也可能是其他应用程序引起的。

这种情形下，可能在最后一帧动画中，你有了设备，但它已经运行了。你必须检测这点，并且可以重获取设备。幸运的是，有一个相当简单的检测它的方法。当你获取一个设备时，检测是否有其他程序在用它。如果是，你只需简单地重获取它，并再次读取数据。因为 `GetDeviceState()` 函数将返回出错代码，所以你可以说出错误所在。

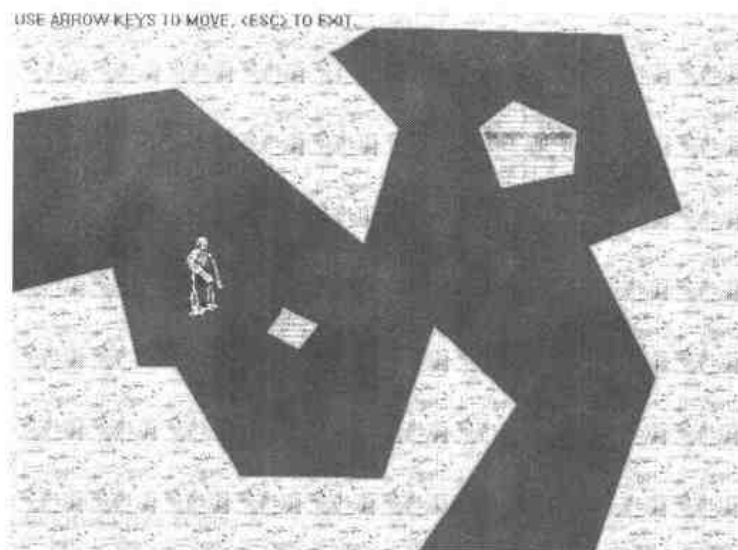


图 9.5 运行 DEMO9_1.EXE

GetDeviceState()返回的实际错误代码如表 9.5 所示。

表 9.5 GetDeviceState() 的错误代码

值	描 述
DIERR_INPUTLOST	设备已经丧失输入功能，并且在下一次调用时将不能请求使用
DIERR_INVALIDPARAM	函数的一个参数无效
DIERR_NOTACQUIRED	已经丢失了全部的设备
DIERR_NOTINITIALIZED	设备未准备好
E_PENDING	数据依然不能获得（真令人寒心）

所以你要做的是在读的过程中检测 DIERR_INPUTLOST，如果有错误就重获取数据。下面是一个例子：

```

HRESULT result; // general result
While(result = lpdikey->GetDeviceState(
    Sizeof(_DIKEYSTATE),
    (LPVOID) keystate) == DIERR_INPUTLOST)
{
    // try an re-acquire the device
    if (FAILED(result = lpdikey->Acquire()))
    {
        break; // serious error
    } // end if
} // end while
    
```

```
// at this point, there is either a serious error or the data is valid
if (FAILED(result))
{ /* error */ }
```

提示

虽然我是在给你重获取键盘的例子，但这发生的几率几乎为零。多数时候，你丢失的将是操纵杆之类的设备。

捕捉鼠标

鼠标是一种用处大得惊人的输入设备。但你能够想像一下那个发明鼠标的家伙吗？你知道有多少人笑话它吗？因为他们觉得它太荒谬了。我祝愿这个发明者在充满珍珠的岛屿上欢笑！就是说，有时候最不寻常的东西工作得最好，鼠标就是一个很好的例子。现在让我们严肃一点……

标准的鼠标有两个或者三个按钮，有 X、Y 两个轴向运动。由于鼠标四处移动，它将描述状态变化的信息打包并连续地(多数时候)发送给 PC。数据被驱动程序获取，最后传送给 Windows 或者 DirectX。至于我们所关心的，鼠标还是黑色之谜。我们想要知道的是如何决定它什么时候移动，什么时候按钮被按下。DirectInput 将完成这些工作，甚至更多。

有两种同鼠标通信的方法：绝对模式和相对模式。在绝对模式下，鼠标返回基于鼠标指针所处的位置的相对于屏幕坐标的位置。因此，在屏幕分辨率为 640×480 时，你所期望的鼠标的位置从 0~639, 0~479 变化。图 9.6 是其示意图。

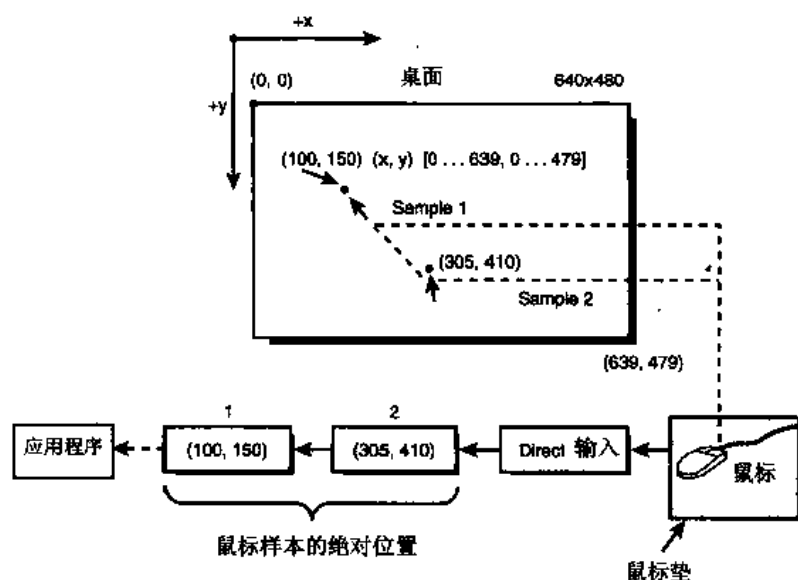


图 9.6 绝对模式下的鼠标

在相对模式下，鼠标驱动程序发送相对于每个时钟的相对值，而不是绝对坐标。如图 9.7 所示。实际上，所有的鼠标都是相对的；是驱动程序在不停地跟踪鼠标的绝对位置。因

此，我喜欢采用鼠标的相对模式，因为它更加灵活。

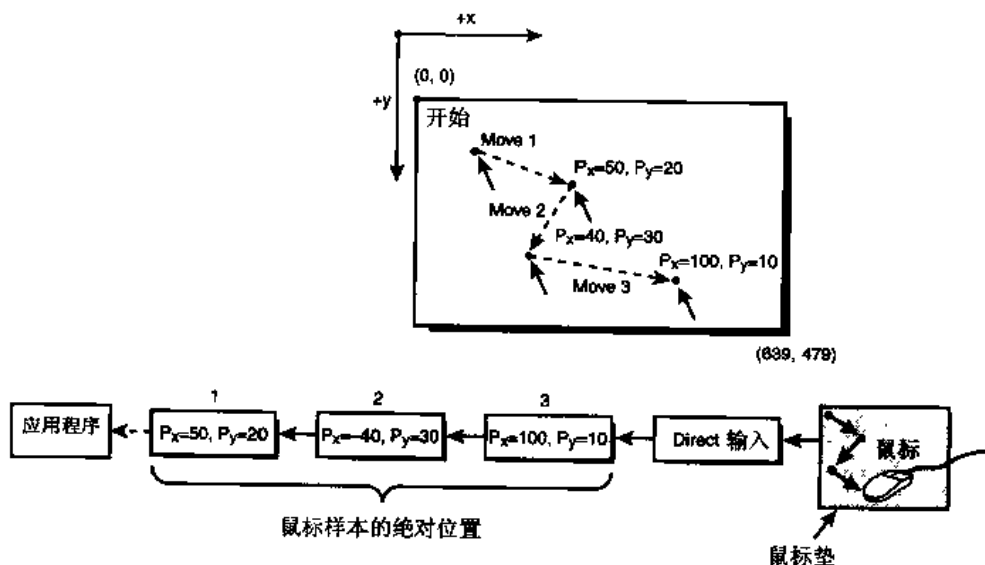


图 9.7 相对模式下的鼠标

现在，既然知道了一些关于鼠标的内容，就让我们看看需要做什么才能够让它在 DirectInput 下工作：

1. 用 CreateDevice() 创建鼠标设备。
2. 用 SetCooperativeLevel() 设置协作等级。
3. 用 SetDataFormat() 设置数据格式。
4. 用 Acquire() 获取鼠标。
5. 用 GetDeviceState() 读鼠标状态。
6. 返回第 5 步直到完成。

注意

如果对这些步骤不熟悉，请参阅前面的键盘部分。

创建鼠标设备

看起来很基础，让我们试试。首先，需要一个接口指针结束创建的指针。像下面这样使用一个 IDirectInputDevice 指针：

```
// of course you need all the other stuff
LPDIRECTINPUTDEVICE lpdimouse = NULL; // the mouse device
// assuming that lpdi is valid
```

```
// create the mouse device
if (FAILED(lpdi->CreateDevice(GUID_SysMouse,
    &lpdimouse, NULL)))
{ /* error */}
```

第一步完成了。注意你使用了 **GUID_SysMouse** 设备常量作为此类型。这将设置一个默认鼠标。

设置鼠标协作等级

现在，设鼠标协作等级：

```
if (FAILED(lpdimouse->SetCooperativeLevel(
    main_window_handle,
    DISCL_BACKGROUND | DISCL_NONEXCLUSIVE)))
{ /* error */}
```

设置数据格式

现在是鼠标数据格式，记住，有一些 **DirectInput** 预先定义的标准数据格式(见表 9.2);你想要的是 **c_dfDIMouse**。将它插入函数并设置数据格式。

```
// set data format
if (FAILED(lpdimouse->SetDataFormat(&c_dfDIMouse)))
{ /* error */}
```

好，现在需要停一下了。使用键盘数据格式 **c_dfDIKeyboard**，数据结构返回一个 256 **UCHARS** 数组。但是，数据格式中定义了一些类似鼠标一样的设备。参见前面的表 9.3。你将采用的数据结构是 **DIMOUSESTATE**，如下所示：

```
// the mouse data structure
typedef struct DIMOUSESTATE
{
    LONG lx; // x-axis
    LONG ly; // y-axis
    LONG lz; // z-axis(wheel in most cases)
    BYTE rgbButtons[4]; // buttons, high bit means down
} DIMOUSESTATE, LPDIMOUSESTATE
```

这样，当你调用 **GetDeviceState()** 获取设备的状态时，就返回这个结构。一点也不奇怪，所有的东西就像它看起来的那样。

获取鼠标

下一步通过调用 `Acquire()` 获取鼠标。就这样：

```
// acquire the mouse
if (FAILED(lpdimouse->Acquire()))
{ /* error */}
```

酷！太简单了，等一会你将所有这些内容打包，就会更加简单。

读鼠标状态

你已经创建了鼠标，设置了协作等级和数据格式，并获取了它。现在可以获得战利品了。在开始，你需要用函数 `GetDeviceState()` 读取数据。但是必须发送一个基于新的数据格式的正确参数 `c_dfDIMouse` 和将要存放数据的数据结构 `DIMOUSESTATE`。这里是怎样读取鼠标：

```
DIMOUSESTATE mousestate; // this holds the mouse data

// ...somewhere in your main loop

// read the mouse state
if (FAILED(lpdimouse->GetDeviceState(sizeof(DIMOUSESTATE)
    (LPVOID)mousestate)))
{ /* error */}
```

技巧



注意这个函数多么潇洒呀！不是采用多个函数，而是使用了一个尺寸和 `ptr` 来对付任何现在存在的或者将来想到的数据格式。这是一个很好的编程技术。

既然有了鼠标数据，让我们使用它吧。假设你想采用鼠标四处移动物体。如果玩家向左移动鼠标一定量，你也想让物体向左移动一定量。另外，如果按下鼠标左键，它就发射火箭，右键退出游戏。下面是主要代码：

```
// obviously you need to do all the other steps...

// defines
#define MOUSE_LEFT_BUTTON 0
#define MOUSE_RIGHT_BUTTON 1
#define MOUSE_MIDDLE_BUTTON 2 // (most of the time)

// globals
```

```

DIMOUSESTATE mousestate; // this holds the mouse data

int object_x = SCREEN_CENTER_X, // place object at center
    object_y = SCREEN_CENTER_Y;

// .. somewhere in your main loop

// read the mouse state
if (FAILED(lpdimouse->GetDeviceState(sizeof(DIMOUSESTATE),
    (LPVOID)mousestate)))
    { /* error */ }

// move object
object_x += mousestate.lX;
object_y += mousestate.lY;

// test for buttons
if (mousestate.rgbButtons[MOUSE_LEFT_BUTTON] & 0x80)
    { /* fire weapon */ }
else
    if (mousestate.rgbButtons[MOUSE_RIGHT_BUTTON] & 0x80)
        { /* send exit message */ }

```

从服务释放鼠标

当你用过鼠标后，首先需要调用函数 `Unacquire()` 反获取它，然后像通常那样释放设备。这里是代码：

```

// unacquire mouse
if (lpdimouse)
    lpdimouse->Unacquire();

// release the mouse
if (lpdimouse)
    lpdimouse->Release();

```

作为一个使用鼠标的例子，我创建了一个叫 `DEMO9_2.CPPIEXE` 的小程序。像以往那样，你需要用 `DINPUT.LIB`、`DDRAW.LIB`、`WINMM.LIB`(C++ 用户)连接，同时还有 `T3DLIB1.CPP`。图 9.8 给出了程序运行时的一个画面。

总之，我想说的就是，对 DirectInput 来说，操纵杆和游戏手柄都是一类东西。它们都有轴、开关、滑杆。仅仅是在操纵杆上的轴有很多(连续)位置，而游戏手柄是固定的或具有极限位置的。关键是每个设备是设备对象、设备物体还是输入对象，依赖于所用的术语和引用名称。它们都仅仅是一些输入设备，正好在一个硬件物体上。懂吗？但愿如此。

除了另外两步，设置类操纵杆设备同鼠标键盘步骤一样。看下面：

1. 用 CreateDevice() 创建操纵杆设备。
2. 用 QueryInterface() 获得指向接口 IDirectInputDevice2 的接口指针(操纵杆需要)。当然，这一步是新加的。
3. 用 SetCooperativeLevel() 设置协作等级。
4. 用 SetDataFormat() 设置数据格式。
5. 用 SetProperties() 设置操纵杆范围、盲区和其他性能。这一步也是新的。
6. 用 Acquire() 获取操纵杆。
7. 用 Poll() 函数轮询操纵杆。这一步是确保在调用 GetDeviceState() 的时候，没有中断驱动程序的操纵杆具有有效数据。
8. 用 GetDeviceState() 读操纵杆的状态。
9. 返回第 8 步直到完成。

枚举操纵杆

我总是不想解释回调函数和枚举函数，因为它们太复杂。但是，从你的手接触到这本书开始，你就需要熟悉这类函数，因为 DOS 编程已经被搁置一些时间了。如果你只是在学习 Windows 编程，这看起来有点困难，但是一旦克服了它，你就无需再担心任何事情了。

一般，回调函数类似于 Windows 程序中的 WinProc()。它是一个你提供的供 Windows 调用的函数。这相当直接且便于理解。图 9.10 给出了类似 Windows 的 WinProc() 那样的标准的回调函数。

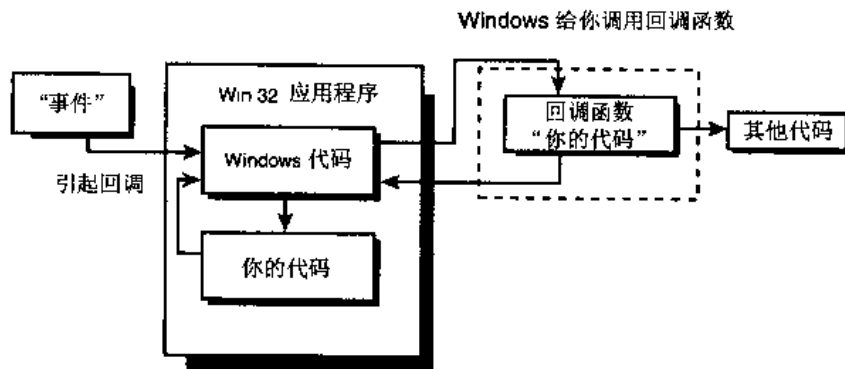


图 9.10 回调函数

但是，Win32/DirectX 在枚举中也使用回调函数。枚举意味着 Windows 需要具有扫描

系统注册表或者其他数据库的能力，查询你要寻找的东西，如插上了什么样的操纵杆和什么样的操纵杆可用。

有两种方法来完成这件事。

- 你可以调用一个 `DirectInput` 函数为你建立一个列表，把它存放在一个数据结构中，你最后分析或推断重要的信息。
- 你可以为 `DirectInput` 提供一个回调/枚举函数，它将调出它所找到的每一个新的设备。你可以使每次回调函数调用，就把新的发现加入你的新建的设备列表中。

第二种方法就是 `DirectInput` 所采用的方法，你只需处理它。现在，你会对需要采用枚举感到迷惑。好，你对插入的操纵杆的类型一无所知，即使你已经知道一点，你还需要准确知道它们中的一个或者多个 `GUID`。所以，无论怎样，都需要扫描它们，因为你需要 `GUID` 来调用 `CreateDevice()`。

完成枚举的函数是 `IDIRECTINPUT::EnumDevice()`，函数从主 `DirectInput` COM 对象直接调用。这里是它的原型：

```
HRESULT EnumDevices(
    DWOED dwDevType, // type of device to scan for
    LPDIENUMCALLBACK lpCallback, // ptr to callback func
    LPVOID pvRef, // 32 bit value passed back to you
    DWORD dwFlags); // type of search to do
```

让我们来看看参数。首先，`dwDevType` 指出了你想要扫描什么样的设备。可能的选项见表 9.6。

表 9.6 DirectInput 的基本设备类型

值	描 述
<code>DIVEVTYPE_MOUSE</code>	鼠标或类鼠标设备（如跟踪球）
<code>DIVEVTYPE_KEYBOARD</code>	键盘或类键盘设备
<code>DIVEVTYPE_JOYSTICK</code>	游戏杆或相似设备，例如方向盘
<code>DIVEVTYPE_DEVICE</code>	不属于上述类别的设备

如果你想让 `EnumDevice()` 识别得更多一些，可以通过对主类型采用逻辑“或”加上一些子类型。表 9.7 给出了一部分鼠标和操纵杆设备枚举的子类型。

表 9.7 DirectInput 子类型（部分）

值	描 述
<code>DIVEVTYPEMOUSE_TOUCHPAD</code>	标准触笔
<code>DIVEVTYPEMOUSE_TRACKBALL</code>	标准跟踪球

续表

值	描 述
DIVEVTYPEJOYSTICK_FLIGHTSTICK	通用飞行杆
DIVEVTYPEJOYSTICK_GAMEPAD	类 Nintendo 游戏键盘
DIVEVTYPEJOYSTICK_RUNDER	简单的方向舵控制
DIVEVTYPEJOYSTICK_WHEEL	方向盘
DIVEVTYPEJOYSTICK_HEADTRACKER	VR 头部追踪仪

注 意



还有一些我没有列出的子类型。关键点在于，DirectInput 可以按照你的要求，总体或者单独搜索。但是，你只需要用 DIDEVTYPE_JOYSTICK 作为 dwDevType 的值，因为你只需要一个基本的、正常运行的操纵杆。

EnumDevice()中的第二个参数是一个指向回调函数的指针。DirectInput 想用它调用它找到的每一个设备。一会我再给你这个函数的格式。下一个参数 pvRef，是一个 32 位指针，指向它要传送给回调函数的值。因此，如果需要，可以改变回调函数中的值，或者用它代替全局变量传回数据。

最后，dwFlag 控制枚举变量如何扫描。也就是，应该扫描的是所有的设备，还是被插入的一些设备，或者只是力反馈设备呢？表 9.8 是一些控制枚举的扫描代码。

表 9.8 控制枚举的扫描代码

值	描 述
DIEDFL_ALLDEVICES	扫描安装的所有设备，即使该设备当前没有连接也不例外
DIEDFL_ATTACHEDONLY	扫描安装和连接的所有设备
DIEDFL_FORCEFEEDBACK	只扫描力反馈设备

警 告



你应该使用 DIEDFL_ATTACHEDONLY，因为它在玩家没有把设备插进计算机时没什么意义。

现在，让我们更仔细地看看回调函数。EnumDevice()工作的方式是，它有一个内部循环，为它找到的每一个设备调用回调函数。如图 9.11 所示。因此，如果你的计算机上有许多插入设备，有可能你的回调函数被调用多次。

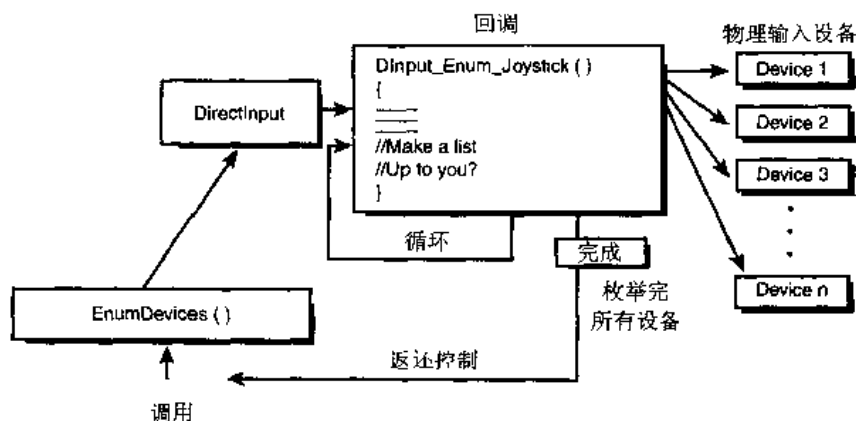


图 9.11 设备枚举流程图

这意味着，你的回调函数将所有的这些设备或者一些东西记录到一张表中，以便在函数 EnumDevice() 返回后，你可以查看它们。酷！记住这点，让我们看一个同 DirectInput 兼容的回调函数原型：

```

BOOL CALLBACK EnumDevsCallback(
    LPDIDEVICEINSTANCE lpddi, // a ptr from DirectInput
                               // containing info about the
                               // device it just found on
                               // this iteration
    LPVOID data); // the ptr sent in pvRef to EnumDevices()

```

你需要完成的就是采用前面的函数原型写一个函数(当然还要写控制代码)，把它作为 lpCallback 传给 EnumDevice()，你的设置就完成了。另外，可以采用任何东西，因为你是通过地址传递函数。

在函数中放置什么由你自己决定，当然，你可能想对所有设备和它们的 GUID 进行记录或者分类，因为它们要被 DirectInput 重载。记住，你的函数对每一个找到的设备调用一次。然后，有了列表在手，你就可以自己或者让玩家从列表中挑选一个，然后使用相关的 GUID 创建设备。

另外，DirectInput 允许你在任何时候持续或者停止枚举。你可以通过回调函数返回值决定。在函数的末尾，你可以返回以下两个常量之一。

DIENUM_CONTINUE——继续枚举。

DIENUM_STOP——停止枚举。

所以如果你简单的在函数中返回 DIENUM_STOP，即使有更多的设备它也只枚举一个。我在这里没有足够的篇幅给出你一个对所有设备的 GUID 进行分类和记录，但是我能够给你一个找到第一个设备并设置它的例子。

上述的枚举函数将枚举第一个设备并停下来。但是，在我把它出示给你之前，快速看

下发送给每次枚举的回调函数的 `DIDEVICEINSTANCE` 数据结构。它充满了关于设备的信息。

```
typedef struct
{
    DWORD dwSize; // the size of the structure
    GUID guidInstance; // instance GUID of the device
                        // this is the GUID we need
    GUID guidProduct; // product GUID of device, general
    DWORD dwDevType; // dev type as listed in tables 9.1-2
    TCHAR tszInstanceName[MAX_PATH]; // generic instance name
                        // of joystick device like "joystick 1"
    TCHAR tszProductName[MAX_PATH]; // product name of device
                        // like "Microsoft Sidewinder Pro"
    GUID guidFFDriver; // GUID for force feedback driver
    WORD wUsagePage; // advanced. Don't worry about it
    WORD wUsage; // advanced. Don't worry about it
} DIDEVICEINSTANCE, *LPDIDEVICEINSTANCE
```

多数时候，唯一感兴趣的字段是 `tszProductName` 和 `guidInstance`。考虑这点，下面是你可以获得第一个操纵杆设备的 `GUID` 的枚举函数：

```
BOOL CALLBACK Dinput_Enum_joysticks(
    LPDIDEVICEINSTANCE lpddi, LPVOID data)
{
    // this function enumerates the joysticks, but stops at the
    // first one and returns the instance guid
    // so we can create it, notice the cast
    (GUID*)guid_ptr = lpddi->guidInstance

    // copy product name into global
    strcpy(joynname, (char *)lpddi->tszProductName);

    // stop enumeration after one iteration
    return(DIENUM_STOP);
} // end Dinput_Enum_joysticks
```

用函数枚举第一个操纵杆，需要这样做：

```
char joynname[80]; // space for joystick name
GUID joystickGUID; // used to hold GUID for joystick

// enumerate attached joystick devices only with
// Dinput_Enum_Joysticks() as the callback function
if (FAILED(lpdi->EnumDevices(
    DIDEVTYPE_JOYSTICK, // joystick only
    Dinput_Enum_Joysticks, // enumeration function
```

```

&joystickGUID, // send guid back in this var
DIEDFL_ATTACHEDONLY)))
{ /* error */}
// notice that we scan for joysticks that are attached only

```

在实际的产品中，你可能想持续枚举，让枚举函数找到所有的设备，在设置或者选择的时候，允许玩家从列表中选择一个，然后你使用那个设备的 GUID 创建设备。下面要谈这个问题。

创建操纵杆

一旦有了想要创建设备的 GUID，通常调用 `CreateDevice()` 来创建设备。假设已经调用了 `EnumDevices()`，并且设备的 GUID 已经被存储在 `joystickGUID` 中了，如何创建操纵杆设备如下所示：

```

LPDIRECTINPUTDEVICE lpdijoy; // joystick device interface
// create the joystick with GUID
if (FAILED(lpdj->CreateDevice(joystickGUID, &lpdijoy,
                             NULL)))
{ /* error */}

```

但是，因为操纵杆需要轮询和其他在 `IDIRECTINPUTDEVICE` 中没有的功能，所以你需要获取 `IDIRECTINPUTDEVICE2` 接口。

获取 IDIRECTINPUTDEVICE2 接口

如果你回忆一下，会记得从 `IUnknown` 继承的所有接口都是基于类的。也就是，它们全部都有 `QueryInterface()` 和 `Release()` 方法。另外，COM 的一个原则是你可以从一个接口得到另一个接口。也就是说，如果你想得到 `IDIRECTINPUTDEVICE2` 接口，你可以用想要获取的 `IDIRECTINPUTDEVICE2` 接口的 IID (Interface ID) 从 `IDIRECTINPUTDEVICE` 接口调用。说得太多了，下面是代码：

```

// version 2 interface pointer
LPDIRECTINPUTDEVICE2 lpdijoy2

// query for the new interface from the old one
lpdijoy->QueryInterface(IID_IdirectInputDevice2,
                        (void **) &lpdijoy2);

```

注意我特地用黑体标出了 **IID**。这不是我编写的，你可以从 DirectX 的头文件中查询它。它现在在 `DINPUT.H` 中，但是，任何人都可以使用它。在任何时候，你都可以从 `DINPUT.H` 中查询你想获取的接口 ID……

现在，既然已经创建了版本 2.0 的操纵杆接口，就无需版本 1.0 的接口了，因此，你可以释放它：

```
if (lpdijoy)
    lpdijoy->Release();
```

从现在开始，只使用 `lpdijoy2` 接口作为你的接口。它具有同样甚至更多的功能。

注 意



在我创建的演示引擎中，为了使用最后的接口，我使用了一个临时指针指向旧的接口。因此，在我的演示程序中，我使用了临时指针，获取最新的一个，通过它调用 `lpdijoy` 而不是 `lpdijoy2`。我这样做是因为我太累了，不想有许多标号的接口。最终结果是在本书中用的接口是可以完成工作的最新的一个。

设置操纵杆的协作等级

设置操纵杆的协作等级同鼠标和键盘一样。但是如果你有一个力反馈杆，你想独占使用它，情况就有所不同。下面是代码：

```
if (FAILED(lpdijoy->SetCooperativeLevel(
    main_window_handle,
    DISCL_BACKGROUND | DISCL_NONEXCLUSIVE)))
{ /* error */}
```

设置数据格式

下面是数据格式，就像鼠标和键盘一样，使用标准的数据格式，见表 9.2。你要用的是 `c_dfDIJoystick` (`c_dfDIJoystick2` 用于力反馈)，如下所示：

```
// set data format
if (FAILED(lpdijoy->GetDataFormat(&c_dfDIJoystick)))
{ /* error */}
```

和设置鼠标相同，需要一定格式的数据结构来保存游戏杆的设备状态数据。参见表 9.3，可以看到，使用的数据结构称为 `DJOYSTATE` (`DJOYSTATE2` 用于力反馈)，如下所示：

```
// generic virtual joystick data structure
typedef struct DIJOYSTATE
{
    LONG lx;    // x-axis of joystick
    LONG ly;    // y-axis of joystick
```

```

LONG lZ;    // z-axis of joystick
LONG lRx;   // x-rotation of joystick(context sensitive)
LONG lRy;   // y-rotation of joystick(context sensitive)
LONG lRz;   // z-rotation of joystick(context sensitive)
LONG rgfSlide[2]; // slider like controls, pedals, etc
DWORD rgdwPov[4]; // Point of View hat controls, up to 4
BYTE rgbButtons[32]; // 32 standard momentary buttons
} DIJOYSTATE, *LPDIJOYSTATE;

```

如你所见, 结构有许多数据段。一般的数据格式是通用的, 我怀疑你是否有必要创建自己的数据结构, 因为我从来没有见过如此之多的操纵杆。无论如何, 注释解释了数据段的意思。轴是有范围的(可以设置), 按钮通常和 0x80(高位)相与, 表示它们被按下。

因此, 当你使用 `GetDeviceState()` 获得设备的状态时, 这个结构将被返回, 也是你要获取的。

快完事了, 还有一个你必须考虑的细节: 那就是在这个结构中将被发送回来的参数值中的细节。一个按钮就是一个按钮。或者按下或者没有, 但是, 制造商则有所不同, 输入范围像 1X、1Y、1Z 可能变化。因此, `DirectInput` 允许你对它们进行缩放, 使它始终满足一个范围, 使你的游戏逻辑数在同样的数字下可以正常工作。让我们看看如何设置这个范围和操纵杆的其他性能。

设置操纵杆的输入性能

因为操纵杆天生是模拟设备, 铀的运动范围是有限的。问题是, 你必须设置一个已知值, 使游戏代码能够解释它。换句话说, 当你获取的操纵杆位置, 它返回 1X=2000, 1Y=-3445, 它是什么意思呢? 你不能够解释它, 因为你没有参考框, 所以让我们来看看你需要分清什么。

至少, 你需要设置你想读取的任何模拟轴的范围(即使它已经数字化了)。例如, 你可以决定把 X、Y 轴分别设成从 -1000~1000 和 -2000~2000。或者两个都设成 -128 ~ 128, 使你能够用一个字节存放。无论你要做什么, 都要采取一定的措施。否则, 重载后, 你没有任何办法解释它, 除非你自己设置了范围。

设置操纵杆的任何性能, 包括操纵杆范围, 都是由 `SetProperty()` 完成的。其原型如下:

```

HRESULT SetProperty(
    REFGUID rguidProp,          // GUID of property to change
    LPCDIPROPHEADER pdiph);    // ptr to property header struct
                                // containing detailed information
                                // relating to the change

```

`SetProperty()` 用于设置一系列性能, 如: 相对或绝对数据格式、每个轴的范围、盲区(或者盲带, 中性区域)等等。使用 `SetProperty()` 函数非常复杂, 因为所有常量本身及嵌套的数据结构就很复杂。

所以说, 除非确有必要, 否则不要调用 `SetProperty()`。多数默认值都能工作得很好。我

就曾浪费了很多时间注视着循环数据结构的工作状况,“故障是什么?”

幸运的是,你只需要设置 X-Y 轴的范围便可使其正常工作,所以,这也是我所要讲解的。如果想学得更多,请参考 DirectX SDK。但是,下面的代码是你用来开始设置其他性能(如果有必要)所需的。需要设置的数据结构如下所示:

```
typedef struct DIPROP_RANGE
{
    DIPROPHEADER diph;
    LONG         lMin;
    LONG         lMax;
} DIPROP_RANGE, *LPDIPROP_RANGE
```

下面是另外一个嵌套结构, DIPROPHEADER:

```
typedef struct DIPROPHEADER
{
    DWORD dwSize;
    DWORD dwHeaderSize;
    DWORD dwObj;
    DWORD dwHow;
} DIPROPHEADER, *LPDIPROPHEADER
```

它们两个都有许多方式进行设置。所以,如果你有兴趣, 请参考 DirectX SDK。你能够发送的标志加起来要超过十页。下面是设置轴的范围所用的代码。

```
// this structure holds the data for the property changes
DIPROP_RANGE joy_axis_range;

// first set x axis to -1024 to 1024
joy_axis_range.lMin = -1024;
joy_axis_range.lMax = 1024;

joy_axis_range.diph.dwSize      = sizeof(DIPROP_RANGE);
joy_axis_range.diph.dwHeaderSize = sizeof(DIPROPHEADER);

// this holds the object you want to change
joy_axis_range.diph.dwObj = DIJOFS_X;

// above can be any of the following:
//DIJOFS_BUTTON(n) - for buttons
//DIJOFS_POV(n) - for point-of-view indicators.
//DIJOFS_RX - for x-axis rotation.
//DIJOFS_RY - for y-axis rotation.
//DIJOFS_RZ - for z-axis rotation (rudder).
//DIJOFS_X - for x-axis.
//DIJOFS_Y - for y-axis.
//DIJOFS_Z - for the z-axis.
//DIJOFS_SLIDER(n) - for any of the sliders.
```

```
// object access method, use this way always
joy_axis_range.diph.dwHow = DIPH_BYOFFSET;

// finally set the property
lpdijoy->SetProperty(DIPROP_RANGE,&joy_axis_range.diph);

// now y-axis
joy_axis_range.lMin = -1024;
joy_axis_range.lMax = 1024;
joy_axis_range.diph.dwSize = sizeof(DIPROPRANGE);
joy_axis_range.diph.dwHeaderSize = sizeof(DIPROPHEADER);
joy_axis_range.diph.dwObj = DIJOFS_Y;
joy_axis_range.diph.dwHow = DIPH_BYOFFSET;

// finally set the property
lpdijoy->SetProperty(DIPROP_RANGE,&joy_axis_range.diph);
```

现在，操纵杆有了 X、Y 轴的范围，是-1024~1024。范围是任意的，但是我喜欢这个范围。注意你所用的数据结构是 DIPROPRANGE。这是你设置参数的结构。最坏的事情就是有太多的方法设置它，实在痛苦。但是采用前面的样板，你至少可以设置任意轴的范围——只需改变 joy_axis_range.diph.dwObj 和 joy_axis_range.diph.dwHow 字段为你想要的值。

作为设置属性的第二个例子，我们看看 X、Y 轴盲区(或者盲带)的设置。盲区是杆中心的中性区域的范围。你有可能想让杆在中心附近时不发送任何数据，如图 9.12 所示。

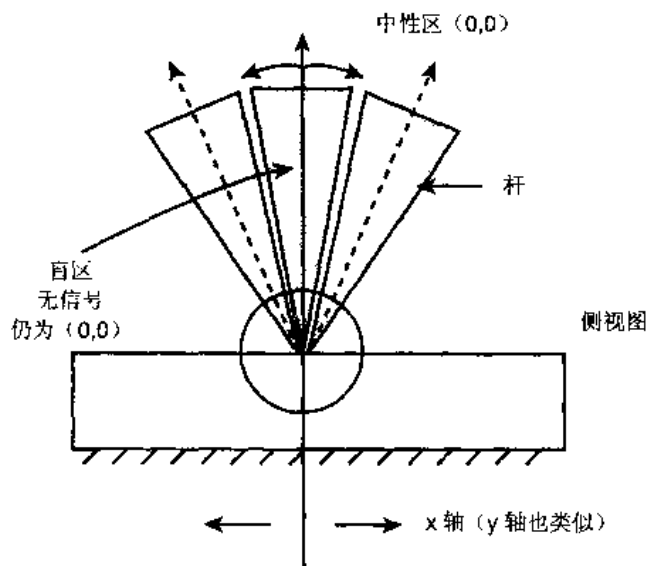


图 9.12 游戏杆盲区

例如，将前面例子中 X、Y 轴的范围设为-1024~1024，如果你想在两个轴上都有 10% 的盲区，就在+、-两个方向取了 102 个单位，对吗？错!!!盲区的值在 0~10000 之间，而不是 1024——10%×10000=1000。这才是你要用的数。

警告



盲区总是 0~10000 之间或者百分数。如果想要盲区为 50%用 5000, 10%则用 1000。

因为这个操作比较简单, 你只需使用 **DIPROPWORD** 结构:

```
typedef struct DIPROPDWOED
{
    DIPROPHEADER diph;
    DWORD        dwData;
} DIPROPDWOED, *LPDIPROPDWOED
```

这要比前面例子中使用的结构 **DIPROP RANGE** 结构简单得多, 如下所示:

```
DIPROPDWOED dead_band; // here's our property word
Dead_band.diph.dwSize    = sizeof (dead_band);
Dead_band.diph.dwHeaderSize = sizeof (dead_band.diph);
Dead_band.diph.dwObj      = DIJOFS_X;
Dead_band.diph.dwHow      = DIPH_BYOFFSET;

// 100 will be used on both sides of the range +/-
Dead_band.dwData = 1000;

// finally set the property
lpdijoy->SetProperty(DIPROP_DEADZONE, &dead_band.diph);

and now for the Y-axis:

Dead_band.diph.dwSize    = sizeof (dead_band);
Dead_band.diph.dwHeaderSize = sizeof (dead_band.diph);
Dead_band.diph.dwObj      = DIJOFS_X;
Dead_band.diph.dwHow      = DIPH_BYOFFSET;

// 100 will be used on both sides of the range +/-
Dead_band.dwData = 1000;

// finally set the property
lpdijoy->SetProperty(DIPROP_DEADZONE, &dead_band.diph);
```

全部就是这样, 感谢上帝。

获取操纵杆

现在, 让我们用 **Aquire()** 获取操纵杆:

```
// acquire the joystick
if (FAILED(lpdijoy->Acquire()))
{ /* error */}
```

当然，要记住在调用 **Release()** 释放设备本身之前，用 **Unacquire()** 反获取操纵杆。

轮询操纵杆

操纵杆是惟一需要轮询的设备(目前为止)。轮询的理由有以下几点：一些操纵杆驱动程序使用中断，数据总是新的。一些驱动程序智能化低(或者说更有效)并且必须轮询。不论驱动程序的哲学观点是什么，你在读取数据之前必须进行轮询。下面是使用的代码：

```
if (FAILED(lpdijoy->Poll()))
{ /* error */}
```

读取操纵杆状态数据

现在已经准备好从操纵杆读取数据了(到现在应该是内行了)。调用 **GetDeviceState()**。但是，你必须发送基于新数据结构的正确参数，**c_dfDIJoystick**(对于力反馈采用 **c_dfDIJoystick2**)，以及存放数据的数据结构 **DIJOYSTATE**。下面是代码：

```
DIJOYSTATE joystate; // this holds the joystick data
// ...somewhere in your main loop

// read the joystick state
if (FAILED(lpdijoy->GetDeviceState(sizeof(DIJOYSTATE)
    (LPVOID)joystate)))
{ /* error */}
```

既然有了操纵杆数据，让我们开始使用它。但是，需要考虑到数据有一定范围。我们来写一个小的程序，到处移动一个物体，就像鼠标的例子那样。并且，玩家如果按下开火键(通常用索引 0)，就发射火箭：

```
// obviously you need to do all the other steps...
// defines
#define JOYSTICK_FIRE_BUTTON 0
// globals
DIJOYSTATE joystate; // this holds the joystick data
int object_x = SCREEN+CENTER_X, // place object at center
    object_x = SCREEN+CENTER_X;
// ... somewhere in your main loop
```

```
// read the joystick state
if (FAILED(lpdijoy->GetDeviceState(sizeof(DIJOYSTATE)
    (LPVOID)joystate)))
    { /* error */}
// move object
// test for buttons
if (mousestate.rgbButtons[JOYSTICK_FIRE_BUTTON] & 0x80)
    { /* fire weapon */}
```

从服务中释放操纵杆

用完操纵杆，需要像通常那样反获取和释放设备。下面是代码：

```
// unacquire joystick
if (lpdijoy)
    lpdijoy->Unacquire();

// release the joystick
if (lpdijoy)
    lpdijoy->Release();
```

警告



反获取前的释放可能是破坏性的。所以一定要先反获取，后释放。

作为一个使用操纵杆的例子，我创建了一个演示程序 DEMO9_3.CPP/EXE。像以前那样，在连接时需要 DDRAW.LIB，WINMM.LIB(VC++用户)，还有 T3DLIB.CPP。图 9.31 给出了程序运行时的一个画面。

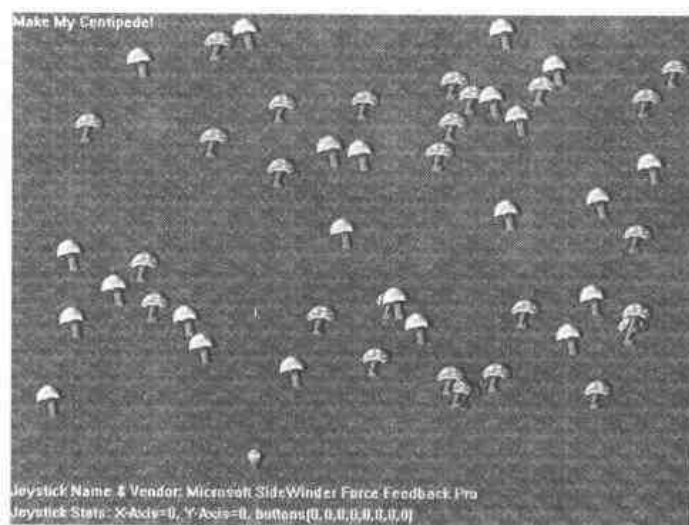


图 9.13 运行 DEMO9_3.EXE

处理输入消息

既然知道了如何读取每个输入设备，问题就变成了输入系统构造了。换句话说，你可能从一系列输入设备上得到了输入，但是，为每个输入设备分类也是很令人头疼的事。因此，你可能会有个好主意，创建一个通用的输入纪录，将从鼠标、键盘、操纵杆所有的输入设备接收的数据合并在一起，然后根据这个结构做出决定。图 9.14 给出了这个概念的图形表示。

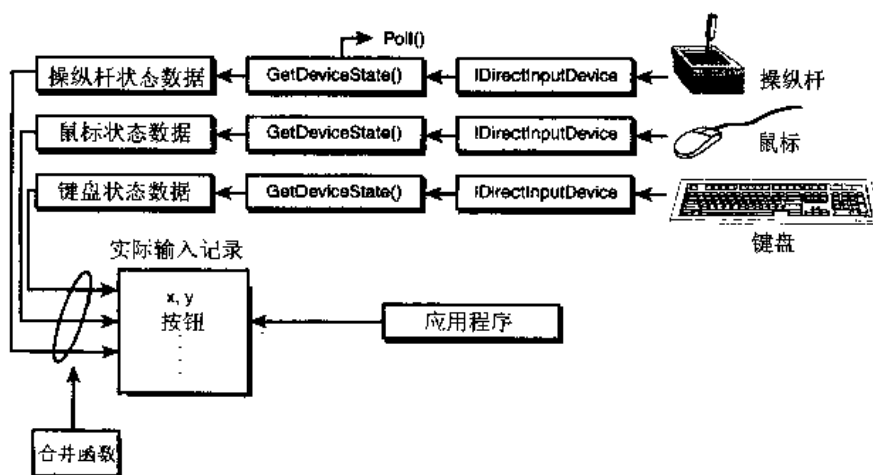


图 9.14 将输入数据合并到一个实际输入纪录

使用简单的老版本 DirectInput 时，我们假设你想让玩家能够同时使用键盘、鼠标、操纵杆进行游戏。鼠标的左键代表开火，键盘使用 Ctrl 键，操纵杆使用的是第一个按钮。另外，如果鼠标向右移动，或者按键盘上右箭头键，或者将游戏杆向右移动，就让所有这些事件均代表玩家向右移动。

作为我所谈内容的一个例子，让我们建立一个简单的输入系统，在系统中，把鼠标、键盘、操纵杆的 DirectInput 输入记录下来，然后把它们合并成为你能够获取的一个记录。这样，在你获取记录时，就无需管它是鼠标、键盘或者操纵杆中哪个的输入，因为所有的输入事件都将被缩放和通用化。

系统要完成的有以下几点：

- X-轴
- Y-轴
- 开火键
- 特殊键

下面是设备变量和事件映像的具体情况。

鼠标映像

+x-轴: 如果($lx > 0$)
 -x-轴: 如果($lx < 0$)
 +y-轴: 如果($ly > 0$)
 -y-轴: 如果($ly < 0$)
 开火键: 左鼠标按钮(`rgbButtons[0]`)
 特殊键: 右鼠标按钮(`rgbButtons[1]`)

键盘映像

+x-轴: 右箭头键
 -x-轴: 左箭头键
 +y-轴: 上箭头键
 -y-轴: 下箭头键
 开火键: Ctrl 键
 特殊键: Esc 键

操纵杆映像(假设两个轴的范围都是-1024~1024, 10%的盲区)

+x-轴: $1X > 32$
 -x-轴: $1X < 32$
 +y-轴: $1Y > 32$
 -y-轴: $1Y < -32$
 开火键: `rgbButtons[0]`
 特殊键: `rgbButtons[1]`

现在已经知道了映像, 下面采用合适的数据结构存放结果:

```
typedef struct INPUT_EVENT_TYP
{
    int dx;    // the change in x
    int dy;    // the change in y
    int fire;  // the fire button
    int special; // the special button
} INPUT_EVENT, *INPUT_EVENT_PTR;
```

使用简单的函数和逻辑, 你将过滤所有的输入到这类结构。首先, 假设你采用以下方式从所有的输入设备重载数据:

```
// keyboard
if (FAILED(lpdikey->GetDeviceState(256,
    (LPVOID)keystate)))
{ /* error */}

// mouse
if (FAILED(lpdimouse->GetDeviceState(sizeof(DIMOUSESTATE),
    (LPVOID)mousestate)))
{ /* error */}

// joystick
if (FAILED(lpdijoy->POLL()))
{ /* error */}
if (FAILED(lpdijoy->GetDeviceState(sizeof(DIJOYSTATE),
    (LPVOID)joystate)))
{ /* error */}
```

此时，有了 `keystate[]` 结构，鼠标、操纵杆是同样的。下面是这项工作的代码；

```
void Merge_Input(INPUT_EVENT_PTR event_data, // the result
    UCHAR *keydata, // keyboard data
    LPDIMOUSESTATE mousedata, // mouse data
    LPDIJOYSTATE joydata) // joystick data
{
    // merge all the data together

    // clear the record to be safe
    memset(event_data, 0, sizeof(INPUT_EVENT));

    // first the fire button
    if (mousedata->rgbButtons[0] || joydata->rgbButtons[0] ||
        keydata[DIK_LCONTROL])
        event_data->fire = 1;

    // now the special button
    if (mousedata->rgbButtons[1] || joydata->rgbButtons[1] ||
        keydata[DIK_ESCAPE])
        event_data->special = 1;

    // now the x-axis
    if (mousedata->lX > 0 || joydata->lX > 32 ||
        keydata[DIK_RIGHT])
        event_data->dx = 1;

    // now the -x-axis
    if (mousedata->lX < 0 || joydata->lX < -32 ||
        keydata[DIK_LEFT])
        event_data->dx = -1;
```



```

// and the y-axis
if (mousedata->lY > 0 || joydata->lY > 32 ||
    keydata[DIK_DOWN])
    event_data->dy = 1;

// now the -y-axis
if (mousedata->lY < 0 || joydata->lY < -32 ||
    keydata[DIK_UP])
    event_data->dy = -1;

} // end Merge_Data

```

当然，你还可以通过检查设备是否在线、缩放数据等等使程序更加复杂，知道这个思想就行了。

力反馈详述

力反馈实际上是一个繁琐的议题。我直到使用时才知道了它的复杂性。我不想涉及到太深的内容。关于力反馈可以写一本书(还有 **DirectMusic**，但是它是另外一件事)。但是，我将给你一个它是什么的基本思想，并且教你设置一个很小的力演示程序。

力反馈描述了下一代输入设备，它们具有激励器、马达等等，可以对手施加力，有时是对整个身体施加力。(“Cybersex”近两年来意思发生了变化)。你可能已经看到或者拥有一个力反馈设备，如微软力反馈操纵杆，或者其他类似设备。

这些设备的编程非常复杂。不但需要对力、弹性、需要的运动有很好的理解，还要充分理解到，设备和力事件或效果同音乐的关系尤其紧密。也就是，它们可以被封装起来，当用于在操纵杆上的不同的马达和激励器时调整力量。因此，一些值，像比率、频率、时间等等在使用和编程中扮有很重要的角色。实际上，创建使用效果或者力反馈命令太复杂，甚至有专门的第三方工具用来创建它们，如微软的 **Force Factory**。幸运的是，你在本演示程序中无需使用任何那类令人迷惑的东西。

力反馈的物理本质

力反馈允许你设置两种类型的效果：运动力和状况。运动力就像总是在流动的活动力，而状况对应于一个事件。每一种情况，你都控制力的大小 N (以牛顿为单位)和力的性能。如方向、持续时间等等。

设置力反馈

创建一个力反馈设备的第一步是找出一个力反馈设备并获得它的 **GUID**。如果你回忆

一下，会记得如何扫描得到标准的操纵杆 GUID，对于力反馈设备需要同样的做法。但是，当你到了设备枚举时，需要这样调用它：

```
GUID fjoystickGUID; //used to hold GUID for forces joystick

// enumerate attached joystick device only with
// Dinput_Enum_Joysticks() as the callback function
if (FAILED(lpdi->EnumDevices(
DIDEVTYPE_JOYDTICK, // joystick only
Dinput_Enum_Joysticks, // enumeration function
&joystickGUID, // send guid back in this var
DIEDFL_ATTACHEDONLY | DIEDFL_FORCEFEEDBACK)))
{ /* error */}
```

一旦有了 GUID，就可以像通常那样创建设备。但是，你必须确保协作等级是 DISCL_EXCLUSIVE 模式(当你使用它的时候，没有其他类型可以使用力反馈)，并且，必须使用版本 2.0 的数据格式。下面是代码：

```
LPDIRECTINPUTDEVICE lpdijoy_temp; // joystick device interface

// version 2 interface pointer
LPDIRECTINPUTDEVICE2 lpdijoy;

// create the joystick with GUID
if (FAILED(lpdi->CreateDevice(joystickGUID, &lpdijoy_temp,
NULL)))
{ /* error */}

// query for the new interface from the old one
lpdijoy_temp->QueryIntf(IID_IDirectInputDevice2,
(void **) &lpdijoy2);

// release the old interface
lpdijoy_temp->Release()
if (FAILED(lpdijoy->SetCooperativeLevel(
main_window_handle,
DISCL_BACKGROUND | DISCL_EXCLUSIVE)))
{ /* error */}

// set data format
if (FAILED(lpdijoy->SetDataFormat(&c_dfDOJoystick2)))
{ /* error */}
```

好了，现在你的力反馈设置完毕，可以使用了。那么，你应该做什么呢？

力反馈演示程序

如果你愿意，可以只将力反馈作为普通的操纵杆使用。但是，现在送过来的数据包是 DJOYSTATE2 而不是 DJOYSTATE。对这段代码的解释需要的时间太长，所以，你可以从它的注释或者演示程序中了解它。

但是，代码一般要设置一个由一个封装和一个过程描述组成的效果。另外，效果同操纵杆开火扳机相连，所以当扣动扳机时它就开始开大，下面是设置力反馈效果的代码，前提是你有设备的 GUID，并且已经像前面那样设置了力反馈操纵杆。

```
// force feedback setup
DWORD      dwAxes[2] = { DIJOFS_X, DIJOFS_Y };
LONG       lDirection[2] = { 0, 0 };

DIPERIODIC diPeriodic;      // type-specific parameters
DIENVELOPE diEnvelope;      // envelope
DIEFFECT    diEffect;        // general parameters

// setup the periodic structure
diPeriodic.dwMagnitude = DI_FFNOMINALMAX;
diPeriodic.lOffset = 0;
diPeriodic.dwPhase = 0;
diPeriodic.dwPeriod = (DWORD) (0.05 * DI_SECONDS);

// set the modulation envelope
diEnvelope.dwSize = sizeof(DIENVELOPE);
diEnvelope.dwAttackLevel = 0;
diEnvelope.dwAttackTime = (DWORD) (0.01 * DI_SECONDS);
diEnvelope.dwFadeLevel = 0;
diEnvelope.dwFadeTime = (DWORD) (3.0 * DI_SECONDS);

// set up the effect structure itself
diEffect.dwSize = sizeof(DIEFFECT);
diEffect.dwFlags = DIEFF_POLAR | DIEFF_OBJECTOFFSETS;
diEffect.dwDuration = (DWORD) INFINITE; // (1 * DI_SECONDS);

// set up details of effect
diEffect.dwSamplePeriod = 0; // = default
diEffect.dwGain = DI_FFNOMINALMAX; // no scaling
diEffect.dwTriggerButton = DIJOFS_BUTTON0; // connect effect
// to trigger button
diEffect.dwTriggerRepeatInterval = 0;
diEffect.cAxes = 2;
diEffect.rgdwAxes = dwAxes;
diEffect.rglDirection = &lDirection[0];
diEffect.lpEnvelope = &diEnvelope;
diEffect.cbTypeSpecificParams = sizeof(diPeriodic);
```

```

diEffect.lpvTypeSpecificParams = &diPeriodic;

// create the effect and get the interface to it
lpdijoy2->CreateEffect(GUID_Square, // standard GUID
    &diEffect, // where the data is
    &lpdieffect, // where to put interface pointer
    NULL); // no aggregation

```

这个演示程序的运行参见 DEMO9_4.CPP。它基本上是采用你的演示程序，并且加了一把机械枪！当然，你需要一个力反馈操纵杆来玩这个游戏。

注 意



前面的力反馈代码基于 DirectX SDK 中的一个例子，所以你可以从那里找到更详尽的解释。

编写通用的输入系统：T3DLIB2.CPP

写一个简单的 `DirectInput` 打包函数几乎毫不费力。可能在某些地方它还是需要一些脑筋，但是多数部分相当简单。你只需要创建一个具有一个非常简单接口和很少参数的 API，参数需要：

- 初始化 `DirectInput` 系统。
- 设置并获取键盘、鼠标、操纵杆(或者任意子设置)。
- 从任意输入设备中读取数据。
- 关闭、反获取、释放每个对象。

我已经创建了一个 API，在 CD 中的 T3DLIB2.CPP.H 中。API 可以为你初始化 `DirectInput` 并读取任何设备。但是，我没有进行输入合并(就像前面例子中那样)。因此，你将以标准的 `DirectInput` 设备状态结构接收输入，并且你将处理每个设备状态结构的不同字段(键盘、鼠标、操纵杆)。但是，这给了你最大的自由度。

在回顾这些函数之前，先看一下图 9.15。图 9.15 给出了各个设备和数据流之间的关系。下面是库函数的全局变量：

```

LPDIRECTINPUT  lpdi;           // dinput object
LPDIRECTINPUTDEVICE  lpdikey;  // dinput keyboard
LPDIRECTINPUTDEVICE  lpdimouse; // dinput mouse
LPDIRECTINPUTDEVICE  lpdijoy;  // dinput joystick
GUID joystickGUID;  // guid for main joystick
char joyname[80];   // name of joystick

```

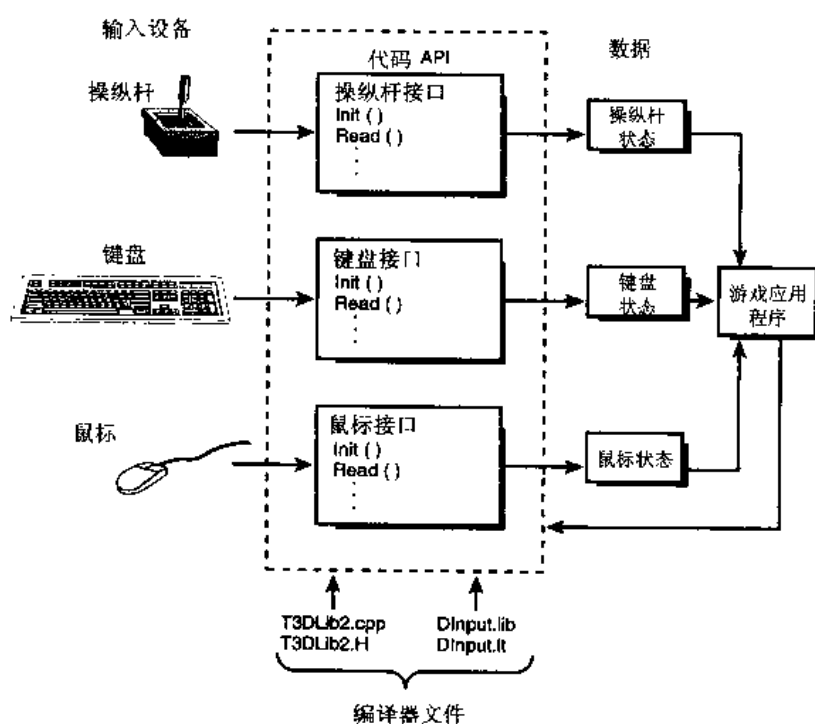


图 9.15 DirectInput 软件系统

```

// all input is stored in these records
UCHAR keyboard_state[256]; // contains keyboard state table
DIMOUSESTATE mouse_state; // contains state of mouse
DIJOYSTATE joy_state;      // contains state of joystick
int joystick_found;        // tracks if stick is plugged in

```

输入系统把从键盘的输入放在 `keyboard_state[]` 中，鼠标数据存放在 `mouse_state` 中，操纵杆数据存放在 `joy_state` 中。这些记录的结构是 DirectInput 的标准设备状态结构。但是，一般情况下，就 X、Y 位置而言，鼠标和操纵杆大致等价。也就是说，可以通过 `IX`、`IY` 和在 `rgbButtons[]` 中的 `BOOLEAN` 按钮访问它。

我们来看看这些函数。变量 `joystick_found` 是一个布尔量，当需要使用操纵杆的时候设置。如果发现操纵杆，它为 `TRUE`，否则为 `FALSE`。你可以用它设计使用操纵杆的代码。不多说了，下面是这个新的 API。

函数原型：

```
int Dinput_INIT(void);
```

目的：

`DInput_Init()` 初始化 DirectInput 输入系统。它创建了主 COM 对象，成功返回 `TRUE`，否则返回 `FALSE`。当然，全局变量 `lpdi` 将变成有效。但是，函数没有创建任何设备。下面是初始化输入系统的一个例子。

```
if (!Dinput_Init())
{ /* error */}
```

函数原型:

```
void Dinput_Shutdown(void)
```

目的:

DInput_Shutdown()释放所有的 COM 对象和在 DInput_Init()调用时分配的资源。一般,你需要在已经释放所有的输入设备本身之后,在应用程序的最后使用 DInput_Shutdown()。下面给一个关闭输入系统的例子:

```
Dinput_Shutdown();
```

函数原型:

```
Dinput_Init_Keyboard(void);
```

目的:

DInput_Init_Keyboard()初始化和获取键盘。它必须一直工作并返回 TRUE,除非是另外一个 DirectX 应用程序以非合作方式在运行。下面是例子:

```
if (!Dinput_Init_Keyboard())
{ /* error */}
```

函数原型:

```
int Dinput_init_Mouse(void);
```

目的:

DInput_Init_Mouse()初始化和获取鼠标。函数无需参数,成功返回 TRUE,失败返回 FALSE。但是,它也应该一直工作,除非是鼠标没有插入或者另外一个 DirectX 应用程序霸占了它。如果运行正常,lpdimouse 将变成有效接口指针。下面是例子:

```
if (!Dinput_Init_Mouse())
{ /* error */}
```

函数原型:

```
int Dinput_Init_Joystick(int min_x=-256, // min x range
                        int max_x=256, // max x range
                        int min_y=-256, // min y range
                        int max_y=256, // max y range
                        int dead_zone=10); // dead zone in percent
```

目的:

DInput_Init_Joystick() 为你初始化操纵杆设备。函数有五个参数，定义了从操纵杆发送的数据的 x、y 范围和一个百分比的盲区。如果你想使用默认的范围 -256~256 和 10% 的盲区，可以不用参数，因为它们是默认值(C++中)。

如果返回了 TRUE，表明操纵杆已经被设置、初始化、获取。调用后，如果需要做其他事情，接口指针 lpdijoy 有效。另外，字符串 joynamex[] 存放一个操纵杆设备的“友”设备名，如微软的 Sidewinder Pro 等等。

下面是一个操纵杆初始化例子，将 x-y 范围设为 -1024~1024，具有 5% 的盲区。

```
if (!Dinput_Init_Joystick(-1024, 1024, -1024, 1024, 5))
{ /* error */}
```

函数原型:

```
void Dinput_Release_Joystick(void);
void Dinput_Release_Mouse(void);
void Dinput_Release_Keyboard(void);
```

目的:

Dinput_Release_Joystick(), Dinput_Release_Mouse() 和 Dinput_Release_Keyboard() 分别释放这些已经使用完毕的设备。即使没有初始化这些设备也可以调用这些函数。所以，如果愿意，可以在应用程序的最后，一起调用它们。下面是一个启动 DirectInput 系统，初始化所有设备，最后释放它们并关闭的复杂例子：

```
// initialize the DirectInput system
Dinput_Init();

// initialize all input devices and acquire them
Dinput_Init_Joystick();
Dinput_Init_Mouse();
Dinput_Init_Keyboard();

// input loop...do work here
// now done...

// first release all devices order is unimportant
Dinput_Release_Joystick();
Dinput_Release_Mouse();
Dinput_Release_Keyboard();

// shutdown DirectInput
Dinput_Shutdown();
```

函数原型:

```
int Dinput_Read_Keyboard(void);
```

目的:

Dinput_Read_Keyboard()扫描键盘,把数据存入 256 字节的 keyboard_state[]。这是一个标准的 DirectInput 键盘的状态数组,所以你如果想利用它,必须使用 DirectInput 键常量 DIK_*。如果键被按下,数组值是 0X80。下面是一个使用 DirectInput 常量单测试右和左箭头键是否被按下的例子(你可以在 SDK 或者在表 9.4 的删节部分中查询这个常量):

```
// read the keyboard
if (!Dinput_Read_Keyboard())
{ /* error */

// now test the start data
if (keyboard_state[DIK_RIGHT]
{ /* move ship right */

else
if (keyboard_state[DIK_LEFT]
{ /* move ship left */
```

函数原型:

```
int Dinput_Read_Mouse(void)
```

目的:

Dinput_Read_Mouse()读取存放在 mouse_state 中鼠标的相对状态,在 DIMOUSESTATE 结构中。数据是相对增量模式。多数时候,你值需要查看 mouse_state.lX、 mouse_state.lY 和存放鼠标三种状态的 rgbButtons[0...2]。下面是读取鼠标并用它移动光标及绘制的例子:

```
// read the mouse
if (!Dinput_Read_Mouse())
{ /* error */

// move cursor
cx+=mouse_state.lX;
cy+=mouse_state.lY;

// test if left button is down
if (mouse_state.rgbButtons[0])
Draw_Pixel(cx, cy, col, buffer, pitch);
```


函数原型:

```
int DInput_Read_Joystick(void);
```

目的:

DInput_Read_Joystick() 轮询操纵杆, 然后读取一个 `DIJOYSTATE` 结构的 `joy_state` 中的数据。当然, 如果没有插入操纵杆, 函数返回一个 `FALSE` 并且 `joy_state` 无效, 但是, 你对此应该清楚。如果成功, `joy_state` 包含操纵杆的状态信息。返回的数据在你前面设置的范围之间, 按钮值以布尔值存放在 `rgbButtons[]` 中。例如, 下面是如何使用操纵杆左右移动一条船的例子, 并且使用第一个按钮开火:

```
// read the joystick data
if (!Dinput_Read_Joystick())
{ /* error */}

// move the ship
ship_x+=joy_state.lx;
ship_y+=joy_state.ly;

// test for trigger
if (joy_state.rgbButtons[0])
    { // fire Weapon }
```

当然, 你的操纵杆可以有許多按钮和多个轴。那种情况下, 你可以使用 `joy_state` 的其他字段, 就像在 `DIJOYSTATE` 中定义的 `DirectInput` 结构那样。

注 意

即使你使用了 `IDIRECTINPUTDEVICE2` 接口, 也没有必要使用 `DIJOYSTATE2` 结构, 因为它只是为力反馈设备设置的。

T3D 库函数一览

现在, 你已经有了组成 T3D 库函数的两个主要的 CPPIH 模型。

- T3DLIB1.CPPIH——DirectDraw 加上图形算法。
- T3DLIB2.CPPIH——DirectInput。

当你编译程序的时候要记住这点。如果你想编译一个程序, 称它为 `DEMOX_Y.CPP`, 然后看它的 `.H` 包含。如果你发现它们包含任意一个 `.H` 模型, 你就需要把 `.CPP` 文件也包含进去。

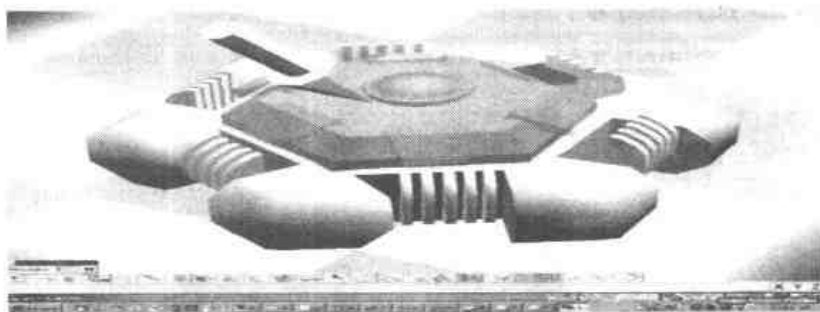
作为一个使用新库函数 T3DLIB2.CPPIH 的例子, 在本章我又写了三个演示程序, 分别把 `DEMO9_1.CPP`、`DEMO9_2.CPP`、`DEMO9_3.CPP` 改写成 `DEMO9_1a.CPP`、`DEMO9_2a.CPP`、`DEMO9_3a.CPP`。所以, 你可以看出, 如果你有了库函数, 有多少代码可以被放弃。

在编译任何一个程序时，请不要忘记包含上源文件的库函数，也不要忘记包含 DirectX.LIB 文件。并且，请把编译器设为 WIN32.EXE。现在，我至少收到了 30 封电子邮件询问设置编译器的问题!我是一个科学家，而不是微软的技术支持!

总结

本章相当有趣，你不这样认为吗?它包含了 DirectInput、键盘、鼠标、操纵杆、输入数据的格式处理、一点力反馈，并为你的数据库添砖增瓦。你学到了 DirectX 用一个通用接口支持所有输入设备，同所有设备通信都只有几步(都很相似)。

情况不错!但是，你还没有走出 DirectX 基本系统的小木屋。下一章将接触到 DirectSound 和 DirectMusic。那之后，你就可以进行一些重要的游戏编程了。



10

用 DirectSound 和 DirectMusic 演奏乐曲

过去在 PC 机上播放声音和音乐简直比登天还难。然而，随着 DirectSound 和 DirectMusic 的出现，这一切都变得很容易了。本章包括下面的内容：

- ➔ 发声原理
- ➔ 数码声音与合成声音
- ➔ 发声硬件
- ➔ DirectSound API
- ➔ 声音文件格式
- ➔ DirectMusic API
- ➔ 对你的库增加声音支持

PC 上的声音编程

编制发声程序是那种被一推再推直到最后才做的事情之一。写一个声音系统的程序很困难，因为你不仅要理解声音和音乐，而且你必须确定声音系统在每一块单独的声卡上都可用。下面是一些相关的问题。过去大多数游戏程序使用第三方声库，例如 Miles 声音系统、Diamondware 声音软件包或相类似的东西。每一个系统都引起人们的争论，但最大的问题是价格。一套 DOS 和 Windows 支持的声库要花数千美元。

现在你不用再担心 DOS 了，但你必须考虑 Windows。Windows 的确支持声音和多媒体，但它却没有被设计为支持高质量的实时视屏游戏的操作系统。值得庆幸的是 DirectSound 和 DirectMusic 解决了所有的问题，并且带给我们更多的方便。DirectSound 和 DirectMusic 不

仅用起来很容易，而且它们在性能上支持上百万种声卡，还可根据你的需要进行扩展。

例如，在 DirectSound3D 下 DirectSound 支持 3D，DirectMusic 则不仅仅能播放 MIDI 文件，它可以完成更多的任务。DirectMusic 是一种新的基于 DLS (Download Sound) 数据的实时音乐调解和录音重放技术。这意味着不仅仅音乐在每一块单独的声卡上发出同样的声音，而且它可以创建出你的基于程序模板图形和你所提供的个性特点的游戏空闲音乐。使用 DirectMusic 来为你的音乐调解需要做很多工作，但在游戏中你想改变游戏者的心情的话，这样做是很值得的，不过你自己也不能把一首歌调解为 10 到 20 个不同的版本。有了这些认识，让我们了解一下声音。

声音产生的原因

声音是那些有清晰定义的物理运动现象之一。如果你走在大街上询问路人什么是声音，他们中的大多数很可能将这样回答，“用你的耳朵听到的东西，如声音或噪音。”（前进一点，再试试……）这样说是对的，但你还是不清楚声音的实际物理特性，如果你要记录控制和发声的话，这是十分重要的。

声音是从声源发出的机械波，如图 10.1 所示。声音只能存在于一种环境中，比如我们的空气中，这里面充满了氮气、氧气和氦气等。声音也能在水中传播，但是其速度要比空气中快得多，因为其传导性随着介质浓度的增加而增加了。

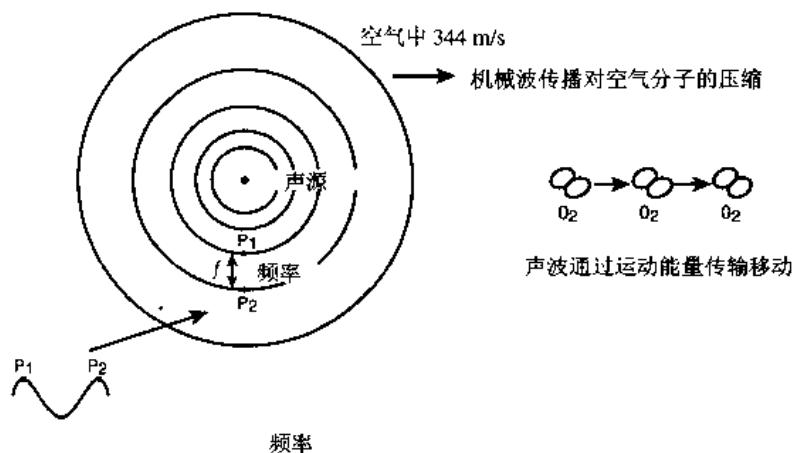


图 10.1 声波

声波实际上是分子的运动。当说话人发言时，它使周围的空气机械地移动，也就是说通过分子的传导在一些点声波传到你的耳朵里。然而，声音的行进是通过空气的机械碰撞传递声波实现的，因而传到你的耳朵要花一些时间。那就是相对而言声音传播得较慢的原因。你可以看到一些事情的发生，例如汽车碰撞时，如果距离足够远的话，直到碰撞发生一或两秒后才听到声音。这是因为机械波或声波在空气中仅仅可以约 600MPH 或 344m/s (米

每秒)的速度传播,具体数值取决于空气的密度和温度。表 10.1 列出了声音在空气、海水和钢铁中的速度,数据适用于通常温度。

表 10.1 声音在不同材料中的速度

材料/介质	声音的近似速度
空气	344m/s
海水	1478m/s
钢铁	5064m/s

如表 10.1 所示,你可以知道为什么水下声纳工程如此之好,但在空气中却不适合(主要是速度太慢)。声纳脉冲或子弹飞行时的声音在水下传播的速度是 1478m/s 或约 $14.78\text{m/s} \times 3.2\text{ft/m} \times 1\text{mi}/5280\text{ft} \times 3600\text{s/h} = 3224.7$ 英里/小时!同空气中平均每小时 750 英里相比,这应该让你清楚为什么在合理的距离以内,声纳扫描对水下移动的物体几乎是瞬间的。

数 学



如果你有兴趣,声音的速度 c (c 不同于 C 和光速 c) 等于频率 \times 波长,或 $f \times \lambda$ 。而且,这个速度可以被计算出来,根据压力、介质浓度,所用方程是:

$$c = \sqrt{\text{压力}/\text{浓度}}$$

这里的压力和浓度是上下文相关的,仅是一个粗略的起点。在现实生活中有许多有关气体、固体和液体的这个方程。

继续说,声音是通过空气以恒速——音速传播的机械波。其中有两个参数——振幅和频率。声波的振幅就是空气移动的数量。一个大声说话的人(或嘴大的人)吞吐了许多空气,所以声强很高。声音的频率就是每秒从发射源发出的全波或循环数,用赫兹或 Hz 来衡量。大多数人可以听到 20~20000Hz 的声音。

而且,男性的噪音声波频率范围是 20~2000Hz,而女性的则是 70~3000Hz。男性噪音较低,女性则较高。图 10.2 是一些标准波形的振幅和频率。

波形可以被认为是声音振幅形状的改变。一些变得很光滑,一些则有突变。即使两个声波有相同的振幅和频率,它们的特别形状也会使我们听起来感觉不一样。

最后,我们用耳朵听到声音,这看起来似乎很简单,但我打算讲讲声音的真正特点。你的耳朵里有一些头发状的称为纤毛的东西。这些纤毛中的每一根都可以探测一个不同范围的声波。当声音像压力脉冲队列一样进入你的耳朵,这些纤毛就会震荡并同相应的声音产生共鸣,并且送一个信号到你的大脑中。然后你的大脑会把这些信号加工成声音意识感觉。然而,在某些星球上生物可以“看”见声音,所以记住,声音这个事物完全是主观的。宇宙中惟一的不变是声音如何传播及声音的物理特性。

总之,声音是展开或收缩的压力波,它在空气中传播。其中收缩率被称为频率,空气被移动的数量被称为相应的振幅或音量。当然,声音有不同的波形,例如正弦波、方波、锯齿波等等。人可以听到的声波范围是 20~20000Hz,普通人的声音则是 2000Hz 左右。不过这些不是声音的全部。

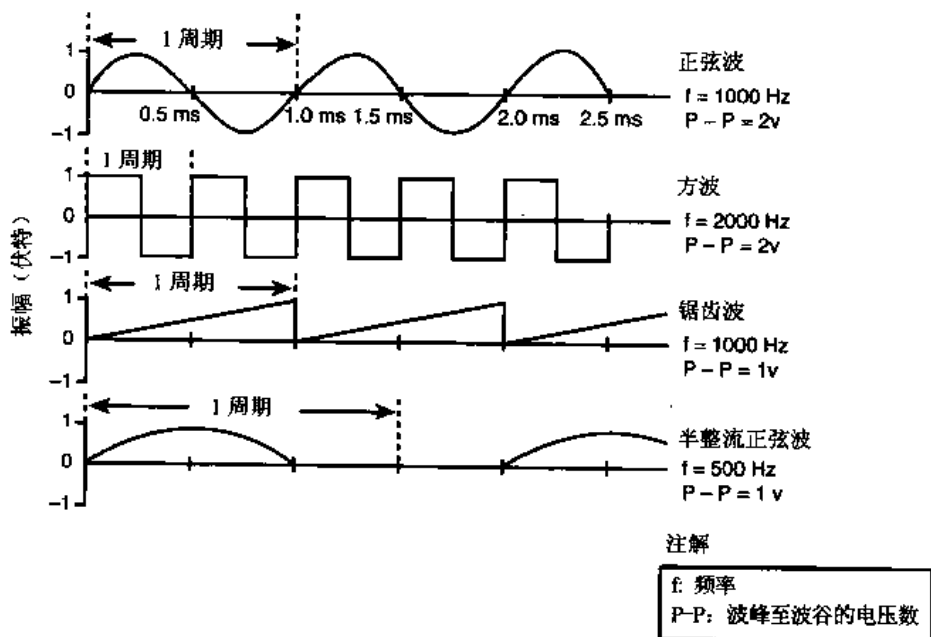


图 10.2 不同的声音波形

一种单独的音调总是正弦波，但它却可以有任意的频率和振幅。单独的音调听起来像是电子玩具或按键式电话(按键式电话中的每个按键发出两种声音，但这足够了)。关键点在于现实世界中大多数声音像噪音、音乐和户外周围的噪音都是由上百种甚至上千种单音混合而成。因此，声音有一个频谱。

数 学



宇宙中的大多数基本波形是正弦波—— $\sin(t)$ 。所有其他的波形都可通过一个线性的组合或一个或多个正弦波的收集而描绘出来。这可通过傅立叶转换而得到数学上的证明，这是把一个波形分解为正弦波结构的方法。同样它也叫我们必须主修头疼的数学课。

声音的频谱是它的频率的分配。图 10.3 显示了男性噪音的频率的分配。你可以看到，男性噪音有许多不同的频率在其中，但多数都较低。关键问题就是要发出实际的真实的声音，你就必须理解声音是由许多不同频率和振幅的单音组成。

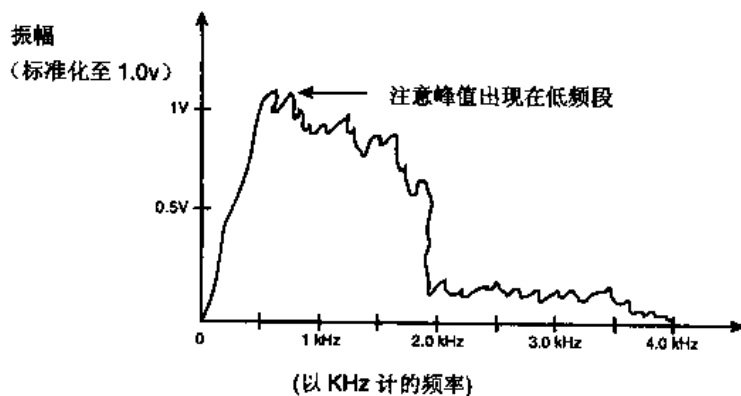


图 10.3 男性噪音的频谱

这些都很重要，但你的目的是要计算机发出声音。没有问题：计算机可以通过电子信号控制扬声器，强迫它以不同的速率、不同的力量(没有原因)发声。让我们看怎样做。

数字与 MIDI—发声大，填充少

计算机可以发出两种类型的声音：数字声音和合成声音。数字声音是声音的基本记录，而合成声音是根据计算算法和硬件发声器重新程序化加工过的声音。数字声音通常用于发声效果上，像爆炸和人们谈话，而合成声音则用于音乐。在最近的大多数日子里，合成声音仅仅用于音乐，并没有用于音效上。然而，返回到 80 年代，程序员们使用 FM 合成器和发声器来制造机车、爆炸、枪炮、打鼓、警报等声音。它们听起来没有数字化的音效，但也工作得很好。

数字声音—从“位”开始

数字化声音包括数字转换，这意味着用 1 或 0 对数据进行编码，例如 110101010110。就象是电子信号可以通过磁场来移动说话者的锥形磁体而发出声音一样，在扬声器中发音会发生相反的效果。那就是扬声器产生一个基于其传感振动的电子信号。这些电子信号有用线性电压或类似编码的声音信息，如图 10.4 所示。

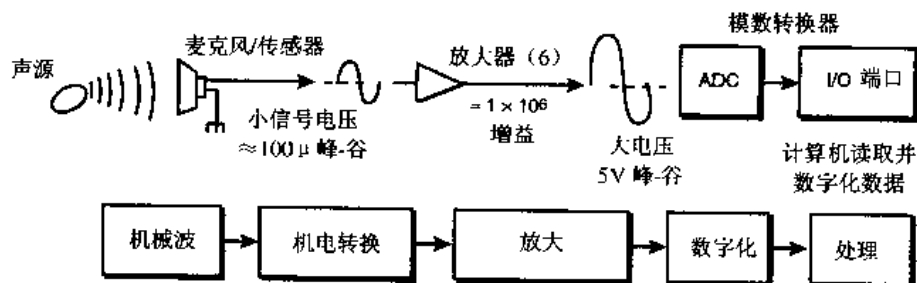


图 10.4 声音的转换

使用正确的硬件，线性电压就可对声音信息进行采样和数字化编码。这就是你的 CD 播放器工作的方式。在 CD 上的信息以数字形式存在，与磁带上的信息很相似。数字信息更容易控制，并且是数字计算机惟一可以加工的信息(很奇怪)。所以用计算机来控制声音，那么该声音必须用类似数字转换器转化为数字数据流，像图 10.5 A 部分所示的那样。

一旦声音被记录进计算机内存中，它就可用数模转换器(D/A)加工或播放，如图 10.5 所示。关键点在于你用计算机发声前要把声音转化为数字格式。但记录数字声音是要有一些技巧的。声音中包含了许多信息。如果你想真实地采样声音，有两个因素你必须考虑：频率和振幅。

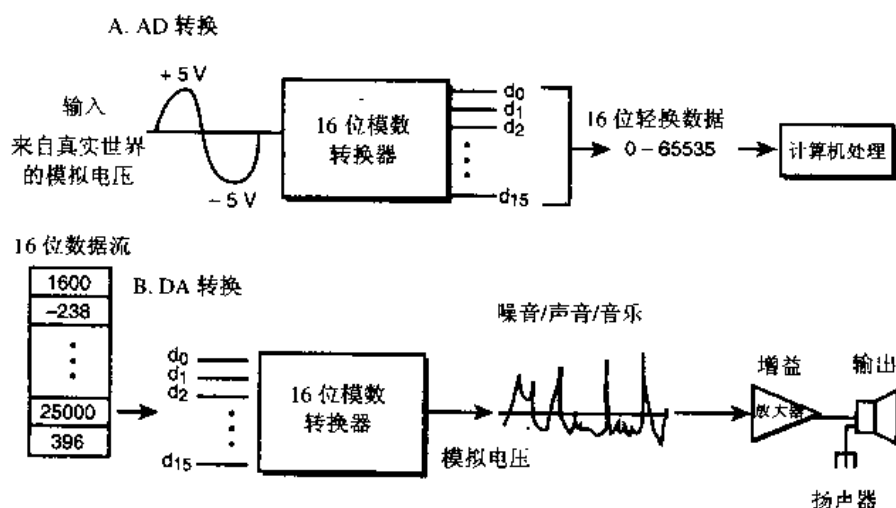


图 10.5 A/D 和 D/A 转换

单位时间内对你所记录声音采样的数量被称为采样率。如果你想真实地再现其原音，那么采样率至少应该是声音频率的两倍。换句话说，如果你正在采样的一个人的噪音的频率为 20~2000Hz，那么你必须以 4000Hz 以上的速率采样。

对这一点的论证是很精确的，也是基于所有的声音都由正弦波构成这一事实。因此，如果你能够采样声音中的最高频率的正弦波，那你就采样声音中所有较低的声音。但要采样频率为 f 的正弦波，你的采样率必须为 $2 \times f$ 。如果仅以 f 的速率采样，你就无法确定是否跟上每个循环的波峰或波谷。换句话说，用两个点来构造正弦波。这被称为 Shannon 定理，最小采样率被称为 Nyquist 频率——它们是相关的吗？

总之，第二个采样参数是振幅分解——即对于振幅有多少不同的值？如果每个采样值你用 8 位表示，那么就有 256 种不同的振幅。对游戏而言这足够了，但要对专门的声音和音乐进行加工的话，你必须用到 16 位，即有 65536 种不同的值。

因此对你来说那是数字声音。主要是对声音的记录或采样使声音从模拟信号转化为数字信号。数字声音对音效和短音是很好的，但对长音的效果并不好，因为它的存储要求是 16 位、44.1KHz，CD 品质的声音是每秒 88KB。换句话说，如果你的游戏是在 CD 上，那么对每个单独的数字音乐你就可以省下 200 个玛戈特。最后，在 99% 的时间里数字音乐比合成音乐好听多了，但在 DirectMusic 下，合成音乐听起来也不错。

合成声音与 MIDI

尽管数字声音是目前最好的声音，合成声音也有很长历史，并正变得越来越好。合成声音不是被数字化记录；它是基于描述的声音的数学再现。合成声音使用硬件和计算算法来从所希望声音的描述中产生。例如，如果你想听 440Hz 的纯音乐会，你可以设计一套能产生频率范围是 0~20000Hz 纯正弦波的硬件，然后让它发出 440Hz 的声音。这是合成声音的基础。

惟一的问题是大多数人并不想听单音（除非你在听一个音乐生日贺卡），所以硬件要同时支持 16~32 种不同的声音，像图 10.6 所示的那样。这并不是坏事，许多不同的视频游戏控制台过去常使用七八十年代的类似东西，但人们仍然不满意。问题在于大多数声音含有许多频率；它们有低音、高音和和声（每种频率的复合）。这使它们听起来清晰饱满。

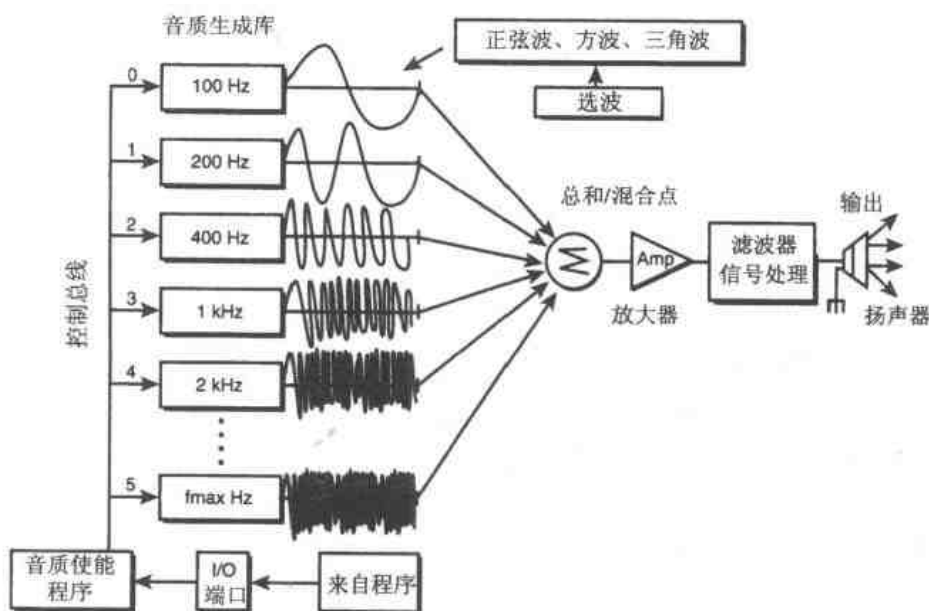


图 10.6 多频声音合成

警告



通常，我不使用“清晰”和“饱满”来描述声音，因为它们不符合我一贯“酷”的作风，但我又不得不使用，这是因为它们是一般的音乐人使用的通用词。所以，请宽容我。

较好声音的首次尝试是 FM 合成。记得 Ad-Lib 卡吗？它是声霸卡的鼻祖和支持复合频道 FM 合成的首张 PC 卡。（FM 代表频率调制。）一个 FM 合成器可以改变的不仅仅是正弦波的振幅，还有频率。

FM 合成操作基于反馈的数学基础。一个 FM 合成器把输出信号反馈给自己，从而调制信号、产生和声并从原始的单正弦波产生同步转换的声音。它们同单音相比听起来逼真得多。

MIDI 概述

几乎同时，FM 合成也出现了，文件格式的音乐合成叫做 MIDI（音乐器具数字接口）。MIDI 是作为时间的函数描述乐曲的语言。MIDI 并不是把声音数字化，而是把它描述为基调、器具和特殊的代码。例如，MIDI 文件可以是这样的：

```
Turn on Channel 1 with a B flat.  
Turn on Channel 2 with a C sharp.  
Turn off Channel 1.  
.  
.  
.  
Turn all channels off.
```

当然，这个信息是以二进制流编码，但你得到了图片。而且，每一个 MIDI 频道都是与不同的器具或声音有关系的。你可以有十六个频道，每一个代表不同的乐器，如小提琴、鼓、吉他、贝司、笛子、喇叭等等。所以 MIDI 是一非直接编码的音乐。

然而，它把合成留给了硬件，仅仅记录实际的音乐音符和时间。在一台计算机上的 MIDI 可能与另一台上的完全不同，这是由于合成的方法与乐器的数据不同。换句话说，可播放一小时的 MIDI 音乐以数字格式存在的话，可以仅仅占用几千字节的内存，而不是要求数兆字节的内存！所以它在很多场合都是很有价值的。

MIDI 和 FM 合成惟一存在的问题是它们仅仅用于音乐时效果才好。当然，你可以设计 FM 合成器来产生用于爆炸或激光束的白噪音，但声音将总是很单调，也不会有数字声音所具备的有机感。因此更多硬件合成的高级方法产生了，例如波表与波导技术。

发声硬件

现在有三类主要的声音合成：FM、波表（也是软件版）和波导。你已经了解了 FM，所以下面让我们看一看波表和波导模型。

波表合成

波表合成是介于合成与数字记录之间的一种混合模式。它是这样工作的：波表中有许多真实的采样的数字声音。这些数据然后被 DSP（数字信号控制器）控制，它获取实际的采样并且以你所需要的频率和振幅来播放。因此，你可以对一个实际的小提琴进行采样，然后使用波表合成来播放同那个小提琴有关的音符。其效果如同数字化的一样，但你仍然要拥有原始的采样。再一次提醒你，那要占用内存。创造性的实验室 AWE32 就是一个很好的例子。

除了硬件波表外，还有基于波表系统的软件合成，例如 MOD 格式和用于 DirectMusic 的 DLS 系统。现在的计算机运行速度非常快，如果你手头上只有一个播放数字声音的数模转换器，你就可以像波表那样用它以软件方式合成对实际乐器的采样值。只要你保证 DSP 实时地工作，并且能够完成对频率、振幅及其他功能的控制，你就不需要任何其他的硬件。DirectMusic 就是这样工作的。

波导合成

波导合成是最终的合成技术。通过 DSP 芯片和专门硬件的使用，声音合成器就可虚拟地产生一个乐器的数学模型，然后简单地播放！这有点像科幻小说，但的确是事实。采用这种技术，人的耳朵就不能察觉采样乐器和波导模拟乐器间的差别。这样你就可以创建控制波表或波导合成器的 MIDI 文件，并且获得更好的结果。创意实验室 AWE64 就有这项技术。

所以结论就是合成器可以产生像实际效果一样的音乐，但音符必须被编码为 MIDI。当然，如果你想有语音或专门的音效，那就必须使用合成器，并且即使用波导技术，你也必须有专门的软件。

然而，用 DirectMusic 你可以用数字化的声音规划乐器，像音符一样来演奏，以使问题得到解决。这样你可用数字声音处理你的音效，用 DirectMusic 处理音乐。当然，除了播放一个波形文件外，还有许多工作要做，但是所有机器上 DirectMusic 发出的同样的声音确是开放的，并且 DirectMusic 可以读取标准的 MIDI 文件，如果使用你会发现有许多特征。因此，你可以决定两者的混合：DirectSound 处理音效，DirectMusic 处理音乐。

数字化记录：工具和技术

在我完成声音和音乐规则程序之前，我想给你一些游戏中记录声音和音乐数据的提示，因为我一下子收到了无数封询问这个问题的 Email。至少有三种数据采样的方式：

- 用麦克风或外部输入从真实世界采样。
- 购买数字或模拟格式的采样声音，下载或记录下来使用。
- 用波形合成器合成数字声音。

第三种方法好像有一些倒退，但如果你想用数字硬件来产生单音，它是很有用的，并且你不用拥有可记录的声源。但对我们来说头两个是最重要的。

如果你正在编制一个有很多话音的游戏，你很可能打算采样你自己的噪音（或朋友的噪音），用一个软件来处理，然后用在你的游戏中。对使用标准的爆炸、通道、咆哮等声音的游戏来说，你很可能放弃一般的声音。

专业声库存在的惟一问题就是价格——一套正宗的专业声库约需 2500 美元。你该怎么办呢？任何计算机存储器都将要一个 5 美元音效的 CD。你必须买几个，但两三个就足以提供你所需要的声音采样——汽车、飞船、怪物等。然而，由于我是一个大方的人，我打算从我的游戏中提供给你一整套不错的声音。它们在该书的配套 CD 的 SOUNDS\目录下。它们都是 WAV 格式，所以你可以在你的游戏中直接使用，但是你也可以重新采样和处理，因为这些声音是从许多游戏产品中拷下来的突变异种的声音。

记录声音

如果要记录你自己的声音，我建议你这样做：每个采样使用 16 位及单一的 22kHz 频率进行。记住，没有立体声效。DirectSound 处理单声是最好的，所以用立体声记录没有什么帮助。大多数你可使用或记录的声音都是单声，因此用立体声记录反而会浪费内存。

如果你要从插入你的声卡的麦克风记录声音，那就要买一个好点的。好的麦克风感觉很沉。正应了那句老话“如果它很沉，它就是好的”。当然，要在没有噪音和干扰的封闭房间里记录你的声音。如果你直接从一个设备记录声音，如 CD 播放器或无线电，确保连接可靠，并使用高质量的音频连接器。

最后，给你的声音文件一个合理的名字。文件名别太模糊了，你不好好组织的话就再也不知道文件是做什么用的。21 世纪了——用长文件名！

处理声音

一旦你用 Sound Forge 或类似的软件对你的声音进行了采样，你很可能想再加工这些声音。当然 Sound Forge 或类似的软件会完成这些工作。在加工时你一定想调准音量，消除杂音及增加回音等。然而，当你作到这一步时你应该备份一下，不要搞乱原来的组织。通过结尾加上数字重新命名加工过的声音文件。否则一旦改变，就无法还原了！

在你加工声音时，实验一下频率转换、回声、失真及其他效果。当你发现一个比较酷的效果时，写下再加工该效果的规则或公式。我可以告诉你，不知道有多少次我从我的声音加工出女性的计算机声音，但我却没有记下来。

最后，完成了所有你的声音之后，以同样的格式写出来，如用 8 位或 16 位，22 kHz 或 11kHz。在你加工声音时这对 DirectSound 是有极大的帮助的。如果你的声音用不同的采样率和位组成加工，DirectSound 将不得不将其转换到本来的 22kHz 8 位格式。

技巧



DirectSound 的原始格式是 22kHz、8 位格式立体声。但自然界中的大多数声音是单声，除非你在不同的位置用两个麦克风记录或有真正的立体声数据，否则对 DirectSound 传送立体声数据是一个浪费。

DirectSound 中的麦克风

DirectSound 是由许多组件或界面组成，就像 DirectDraw 一样。然而，这本书的主要内容在于游戏编程，所以我们把时间放在重点部分。因此，我不打算讨论 3D 声音组件、DirectSound3D 或声音捕获界面及 DirectSoundCapture。我打算把讨论焦点放在 DirectSound 的主画面上。相信我，那足以让你忙个不休。

图 10.7 说明了 DirectSound 与 Windows 子系统的关系。你可以注意到它与 DirectDraw 非常相似。然而 DirectSound 却有一个 DirectDraw 所没有的很酷的特征——即使没有一个支持声卡的 DirectSound 的驱动程序，DirectSound 仍然可以工作，但它使用的是仿真和 Windows DDI。因此只要你用 DirectSound 装载动态连接库 DLL，即使没有 DirectSound 支持的声卡驱动，你的代码也一样可以运行。它运行速度可能不会太快，但它确实可以运行。这就是它酷的地方。

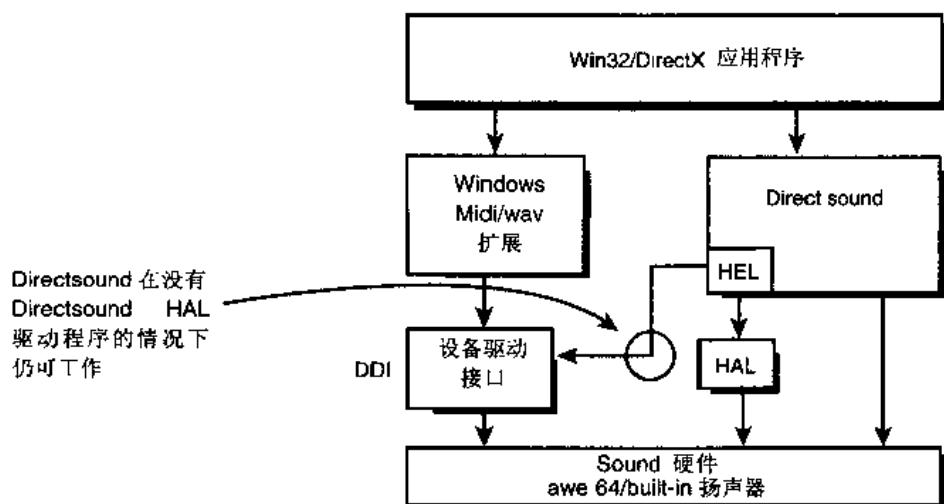


图 10.7 DirectSound 在 Windows 中的位置

就我们关心的而言 DirectSound 有两个组件：

- 当你使用 DirectSound 时加载的运行 DLL。
- 编译时的库文件 DSOUND.LIB 和头文件 DSOUND.H。

要创建一个 DirectSound 应用，所有你需要做的就是你的应用代码中包含进这两个文件，所有的问题都会得到很好的解决。

要使用 DirectSound，你必须创建一个 DirectSound COM 对象，然后从主对象中申请各种界面。图 10.8 说明了 DirectSound 的主界面。

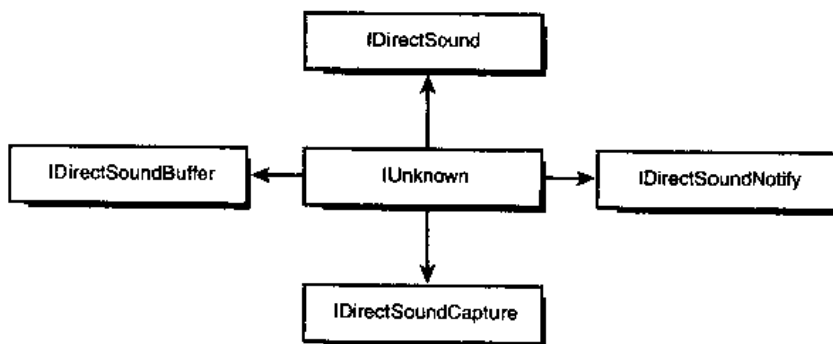


图 10.8 DirectSound 的界面

- **IUnknown**——所有 COM 对象的基本 COM 对象。

- **IDirectSound**——DirectSound 的主 COM 对象。它代表音频硬件本身。如果你的计算机中有一张或多张声卡，那么每一张都需要一个 DirectSound 对象。
- **IDirectSoundBuffer**——代表混合硬件和实际的声音数据。有两种类型的 DirectSound 缓冲：主缓冲和辅助缓冲（看看 DirectSound 是如何同 DirectDraw 相似的）。只有一个主缓冲代表正在播放的声音，并且是通过硬件或软件把它们结合起来。辅助缓冲代表录音重放的存储声音。它们可以存在系统内存或声卡的 SRAM（声音 RAM）中。另一种情况下，只要有足够的马力和内存你就可以播放尽可能多的辅助缓冲中的声音。图 10.9 代表主声音缓冲和辅助声音缓冲的关系。

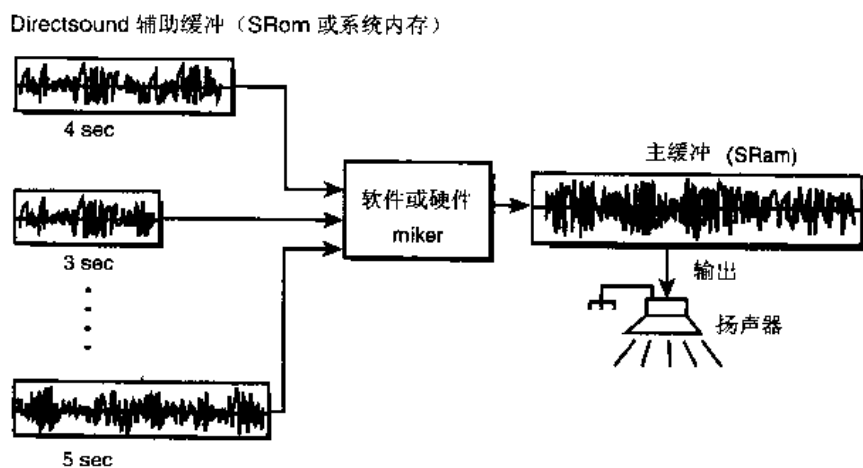


图 10.9 声音缓冲

- **IDirectSoundCapture**——你不打算使用这个界面，但像我所说的那样，它在记录和捕获声音时是有用的。你可以用它让播放器记住它的名字，或如果想更巧妙的话，可用于捕获声音识别的实时演说。
- **IDirectSoundNotify**——这个界面被用于传送消息给 DirectSound。在游戏中你可能需要它处理复杂的声音系统，但没有它也可处理。

要使用 DirectSound，你要首先创建主 DirectSound 对象，创建一个或多个辅助声音缓冲，用声音文件加载然后播放你想播放的东西。DirectSound 会处理所有的细节，如和音。因此让我们从创建主 DirectSound 对象开始。

启动 DirectSound

主 DirectSound 对象代表一张声卡。如果你有多张声卡，就必须计算检测和申请它们的 GUID（全局惟一标识）。但如果你仅仅想同默认声音设备连接，你就不需要用检测的方法来混用；你可以简单地创建一个代表主声卡对象的 DirectSound 对象：

```
LPDIRECTSOUND lpds; //directsound interface pointer
```

要创建一个 **DirectSound** 对象，你必须调用 **DirectSoundCreate()**，其原型如下：

```
HRESULT DirectSoundCreate(
    LPGUID lpGuid, //guid of sound card
    // NULL for default device
    LPDIRECTSOUND *lpDS, //interface ptr to object
    IUnknown FAR * pUnkOuter) //always NULL
```

调用过程与过去创建 **DirectDraw** 对象非常相似。通常这些资源都很相似；一旦你掌握了 **DirectX** 中的一部分，你就已经掌握了它们全部。问题在于微软不停地增加界面，你能学多快他们就加多快！总之，要创建一个 **DirectSound** 对象，要这样做：

```
LPDIRECTSOUND lpds; //pointer to directsound object
//create DirectSound object
if (DirectSoundCreate(NULL, &lpds, NULL) != DS_OK)
    { /* error */ }
```

注意到调用成功的返回值是 **DS_OK** (**DirectSound OK**) 而不是 **DD_OK** (**DirectDraw OK**)。然而，那仅是一个告诉你成功调用的一个例子的代码。用 **FAILURE()** 和 **SUCCESS()** 宏来检查成功与失败，就像下面这样：

```
//create DirectSound object
if (FAILED(DirectSoundCreate(NULL, &lpds, NULL)))
    { /* error */ }
```

当然，当你使用完 **DirectSound** 对象之后，你必须这样释放它：

```
lpds->Release();
```

这一步是在卸载你的应用时执行。

理解协作水平

在你创建了主 **DirectSound** 对象之后就应设定 **DirectSound** 的协作水平。就协作水平而言 **DirectSound** 比 **DirectDraw** 更有技巧性。在你接管声音系统时你不能像用图表算法那样太随意。如果你一定要这样做，那随你便了，但微软建议你不要任性，因此最好接受他们的建议。

有许多 **DirectSound** 可以设定的协作水平。可以分成两类：提供可控制主声音缓冲功能的设定和非控制声音缓冲功能的设定。记住，主缓冲代表实际的和声硬件或软件，它始终在工作，传送数据给扬声器。如果你混合使用主缓冲，**DirectSound** 将要求你确定你知道你在做什么，因为它不仅可以破坏你的应用声音文件，也会毁掉其他人的东西。下面是每个协作水平的简短解释：

- **Normal Cooperation**——这是所有设定中最主要的。当你的应用有焦点时，它就可以播放声音，但不妨碍其他应用。而且你不需要向主缓冲写入命令，DirectSound就会为你创建一个隐含的 22kHz、立体声、8 位的默认主缓冲。我建议你始终使用该设定。
- **Priority Cooperation**——用该设定，你就有了进入所有硬件的入口，你可以改变主混频器的设定，你可以要求硬件完成高级内存操作，如压缩。这个设定只有在你必须改变主缓冲的数据格式时才是必要的——比如你想播放 16 位的采样值时可以做。
- **Exclusive Cooperation**——同 Priority 一样，但只有你的应用在前台时才听得见。
- **Write_Primary Cooperation**——这是最高优先级。要听到声音，你就要完全控制并且必须自己控制主缓冲。如果你曾写你自己的声音混频器或引擎你就可只使用该模型——我想只有 John Miles 这样做。

设置协作水平

在我看来，在你掌握 DirectSound 的用法以前你应该使用普通优先级。它是最容易工作和操作的。要设定协作水平就从主 DirectSound 对象中使用 SetCooperativeLevel() 函数。下面是原型：

```
HRESULT SetCooperativeLevel(HWND hwnd,           //window handle
                             DWORD dwLevel);      //cooperation level setting
```

如果调用成功的话则返回 DS_OK，否则返回其他值。但记住检测错误，因为很可能别的应用在控制声卡。表 10.2 列出了不同协作水平的不同设定标志。

表 10.2 DirectSound 中 SetCooperativeLevel() 的设置

值	说 明
DSSCL_NORMAL	普通水平
DSSCL_PRIORITY	优先水平，允许你设定主缓冲的数据格式
DSSCL_EXCLUSIVE	当你的应用在前台时除了排斥控制之外为优先水平
DSSCL_WRITEPRIMARY	完全控制主缓冲

下面是在你创建完 DirectSound 对象之后如何设定协作水平：

```
if (FAILED(lpds->SetCooperativeLevel(main_window_handle,
                                     DSSCL_NORMAL)))
    { /* error setting cooperation level */ }
```


很酷，不是吗？看一看 CD 中的 EDMO10.CPP1EXE。它创建了一个 DirectSound 对象，设定了协作水平，然后在退出时释放了对象。它并没有发出任何声音——那是下面的内容！

技巧



当你从这一章编译程序时，确保在你的工程中包含进了 DSOUND.LIB。

主要与辅助的声音缓冲

代表本身的 DirectSound 对象单独有一个主缓冲。该主缓冲代表声卡上的混频硬件（或软件）并始终运行，就如同一个小传送带。手动主缓冲混频十分复杂，所幸你不必自己完成，只要你没有将协作水平设为最高优先级，DirectSound 就会为你照顾好主缓冲。另外，只要你把协作水平的优先级设定得低一些，如 DSSCL_NORMAL，DirectSound 就会为你创建一个主缓冲而不需要你自己创建。

惟一缺憾的是主缓冲将被设定为 8 位表示的 22kHz 立体声。如果你想用 16 位或更高的回放率，你就得至少将协作水平设为 DSSCL_PRIORITY，然后为主缓冲设置新的数据格式。但现在只需使用默认值。

使用辅助缓冲

辅助缓冲代表你想播放的声音。只要你的计算机有足够的内存它们可以为任意大小。然而，声卡上的 SRAM 却只能存储一定的数据，因此当在声卡上存储声音数据时一定要当心。但存储在声卡上的声音数据只消耗很少的能量就可以控制，所以要把这一点记在心里。

现在有两种类型的辅助缓冲——静态和动态。静态声音缓冲中是你计划保存及要反复播放的声音。它们是 SRAM 或系统内存很好的替代品。动态声音内存则稍有不同。假设你准备用 DirectSound 播放一整张 CD，我认为你可能没有太多的系统 RAM 或 SRAM 来存储 650M 的音频数据，因此你不得不大块地读取数据并且送到 DirectSound 缓冲里。这就是为什么要用动态缓冲的原因。你不断地用要播放的新声音数据来填充替换动态缓冲。很有趣吧？看一看图 10.10。

通常，所有的辅助声音缓冲都可以被写入静态或动态缓冲中。然而，由于声音在你写入时可能正在播放，DirectSound 就用了 Circular buffering 来对此加以计划，这意味着每个声音数据都是以循环数组的形式存储，也即通过一个播放指针不断的从一个点读取数据，通过另一个指针写到另一个点。如果你不需要在声音被播放时将数据写到你的缓冲里，你就不必担心这一点，但在你输出声音流时你是要考虑的。

为使问题简单化，声音缓冲的数据访问函数可能返回一个被分成两部分的内存地址，因为你试图写入的数据块存在于缓冲的末端，溢出部分则流入缓冲始端。如果你要输出声

音流，你就需要清楚这一点。然而，这在大多数游戏中是没有意义的，因为只要保持几秒钟声效同时按要求加载所有的音乐，你就可以用儿兆的 RAM 存储所有的数据。在 32MB 以上的计算机中使用 2~4MB 的存储空间储存声音没有什么问题。

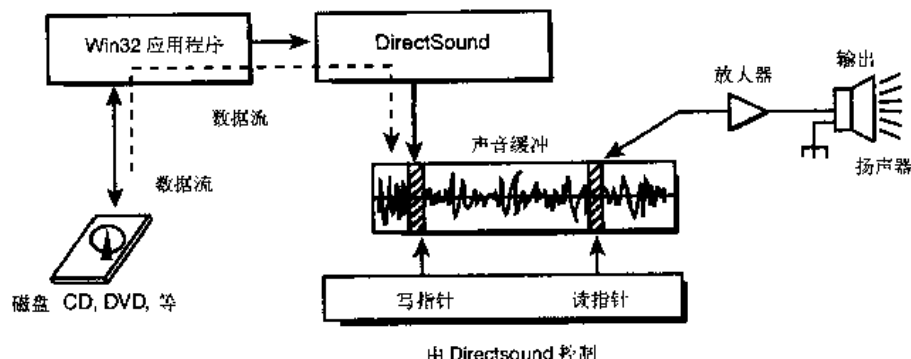


图 10.10 输出音频数据

创建辅助声音缓冲

要创建辅助声音缓冲，你必须用正确的参数调用 `CreateSoundBuffer()`。如果调用成功，函数就会创建一个声音缓冲，对其进行初始化后返回一个如下类型的画面指针：

```
LPDIRECTSOUNDBUFFER lpdsbuffer; //size of this structure
```

然而，在你调用 `CreateSoundBuffer()` 之前，你必须建立一个 `DirectSoundBuffer` 的描述结构，它与 `DirectDrawSurface` 的描述结构很相似。该描述结构是 `DSBUFFERDESC` 类型，下面是它的原型：

```
typedef struct
{
    DWORD dwSize;          //size of this structure
    DWORD dwFlags;         //control flags
    DWORD dwBufferBytes;   //size of the sound buffer in bytes
    DWORD dwReserved;      //unused
    LPWAVEFORMATEX lpwfxFormat; //the wave format
}DSBUFFERDESC, *LPDSBUFFERDESC;
```

`dwSize` 是 DirectX 的标准大小，`dwBufferBytes` 是你所希望的缓冲区的大小，`dwReserved` 没有使用。真正有趣的字段是 `dwFlags` 和 `lpwfxFormat.dwFlags`，包含着声音缓冲的创建标志。看一看表 10.3，其中列出更多的基本标志设定。

表 10.3 DirectSound 辅助缓冲创建标志

值	说 明
DSBCAPS_CTRLALL	缓冲拥有全部控制功能
DSBCAPS_CTRLDEFAULT	缓冲拥有隐含的控制功能。同标志 DSBCAPS_CTRLPAN、DSBCAPS_CTRLVOLUME 及 DSBCAPS_CTRLFREQUENCY 的说明一样
DSBCAPS_CTRLFREQUENCY	缓冲拥有频率控制功能
DSBCAPS_CTRLPAN	缓冲拥有总控制功能
DSBCAPS_CTRLVOLUME	缓冲拥有音量控制功能
DSBCAPS_STATIC	表明缓冲用于静态声音数据，如果可能的话，你将在硬件内存中创建这些缓冲
DSBCAPS_LOCHARDWARE	如果内存足够的话使用硬件混频和内存建立声音缓冲
DSBCAPS_LOCSOFTWARE	迫使缓冲存储在软件内存中，并且使用软件混频，即使 DSBCAPS_STATIC 被标识及硬件资源可用
DSBCAPS_PRIMARYBUFFER	表明缓冲是主声音缓冲。如果你想创建主缓冲同时你是音乐发烧友的话，只需要设定该标识即可

通常情况下你为默认控制、静态声音及系统内存分别设定 DSBCAPS_CTRLDEFAULT、DSBCAPS_STATIC 及 DSBCAPS_LOCSOFTWARE 就可以了。如果你想使用硬件内存，用 DSBCAPS_LOCHARDWARE 代替 DSBCAPS_LOCSOFTWARE。

说 明



你对声音的控制功能越多，那么在听到声音前停顿（软件过滤）就越多。这意味着控制时间增多。当然，如果你不需要音量控制、完全音效、频率转换，那么只要设定 DSBCAPS_CTRLDEFAULT 并仅使用完全功能就可以了。

现在让我们看看 WAVEFORMATEX 结构。它是一个代表你希望使用的缓冲的声音文件描述（也是标准的 Win32 结构），其中的参数像录音重放率、频道数量（单频或双立体声），采样量的位数等都记录在该结构中。下面是它的原型：

```
typedef struct
{
    WORD_ wFormatTag;           //always WAVE_FORMAT_PCM
    WORD_ nChannels;            //number of audio channel 1 or 2
    DWORD nSamplesPerSec;       //samples per second
    DWORD nAvgBytesPerSec;      //average data rate
    WORD nBlockAlign;           //nchannels * bytespersmple
    WORD nBitsPerSample;        //bits per sample
    WORD cbSize;                //advanced, set to 0
}WAVEFORMATEX;
```

很简单吧。WAVEFORMATEX 最基本的功能就是对声音的描述，同时，你需要像 DSBUFFERDESC 部分那样建立一个实例。让我们从 CreateSoundBuffer()函数的原型开始看看怎样做：

```
HRESULT CreateSoundBuffer(
    LPDIRECTSOUNDBUFFER lpDSBufferDesc,    //ptr to DSBUFFERDESC
    LPDIRECTSOUNDBUFFER lpDSBuff,          //ptr to sound buffer
    LPUNKNOWN FAR *pUnkOuter);              //always NULL
```

下面是以 11kHz、8 位模式建立的一个辅助 DirectSound 缓冲的例子，该例子足以存储两秒钟的数据：

```
//ptr to directsound
LPDIRECTSOUNDBUFFER lpdsbuffer; buffer
DSBUFFERDESC dsbd;           //directsound buffer description
WAVEFORMATEX pcmwf;          //holds the format description
//set up the format data structure
memset(&pcmwf, 0, sizeof(WAVEFORMATEX));
pcmwf.wFormatTag = WAVE_FORMAT_PCM; //always need this
pcmwf.nChannels = 1;                //MONO, so channels=1
pcmwf.nSamplesPerSec = 11025;        //sample rate 11khz
pcmwf.nBlockAlign = 1;              //see below
//set to the total data per
//block, in our case 1 channel times 1 byte per sample
//so 1 byte total, if it was stereo then it would be
// 2 and if stereo and 16 bit then it would be 4
pcmwf.nAvgBytesPerSec =
    pcmwf.nSamplesPerSec * pcmwf.nBlockAlign;
pcmwf.wBitsPerSample=8; //8 bits per sample
pcmwf.cbSize = 0; // always 0
// set up the directsound buffer description
memset(&dsbd, 0, sizeof(DSBUFFERDESC));
dsbd.dwSize=sizeof(DSBUFFERDESC);
dsbd.dwFlags=DSBCAPS_CTRLDEFAULT | DSBCAPS_STATIC |
    DSBCAPS_LOCSOFTWARE;
dsbd.dwBufferBytes = 22050; //enough for 2 seconds at
    // a sample rate of 11025
dsbd.lpwfxFormat = &pcmwf; //the WAVEFORMATEX struct
// create the buffer
if(FAILED(lpds->CreateSoundBuffer(&dsbd, &lpdsbuffer, NULL)))
    { /* error */ }
```

如果函数调用成功的话，一个新的声音缓冲就建立了，其中的数据就是准备播放的。唯一的问题在于缓冲区中并没有任何数据，你必须用你的声音数据填充声音缓冲，你可以读取一个.VOC、WAV、AV 或其他的声音数据文件而得到。你也可以创建计算算法数据然后将其写入缓冲区中。让我们看看如何把数据写入缓冲区中，稍后我将介绍如何从盘上

读取声音文件。

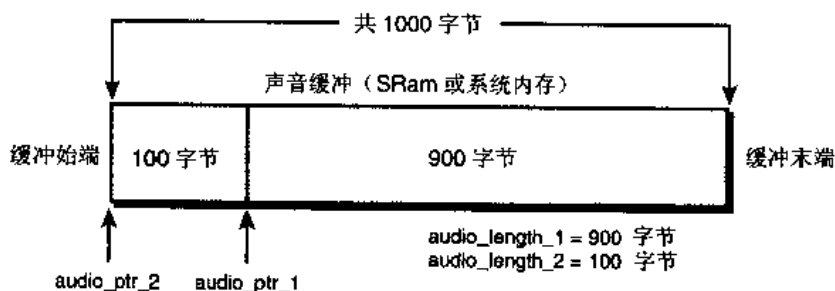
把数据写入声音缓冲

正如我所指出的，辅助缓冲是自然循环的，因此写入数据比处理标准的线性数组数据要复杂些。例如，通过使用 DirectDraw 界面，你仅仅锁定了表面内存而向其中写入数据。(这是惟一的可能，因为有一个程序会将非线性内存转换为线性内存。)DirectSound 以相似的风格工作：先锁定它，但不是得到一个返回指针，而是两个！因此，你必须先把一部分数据写入第一个指针指向的内存，其余的写入第二个指针指向的区域。看一下 Lock() 的原型来理解我的意思。

```
HRESULT Lock(
    DWORD dwWriteCursor,           //position of write cursor
    DWORD dwWriteBytes,           //size you want to lock
    LPVOID lplpvAudioPtr1,        //ret ptr to first chunk
    LPDWORD lpdwAudioBytes1,       //num bytes in first chunk
    LPVOID lplpvAudioPtr2,        //ret ptr to second chunk
    LPDWORD lpwAudioBytes2,       //num of bytes in second chunk
    DWORD dwFlags);               //how to lock it
```

如果你把 dwFlags 设定为 DSBLOCK_FORMWRITECURSOR，那么缓冲将从当前缓冲写入点被锁住。如果你把 dwFlags 设定为 DSBLOCK_ENTIREBUFFER，缓冲将被完全锁住。这就是解决问题的思路。始终很简单。

例如，创建一个容量为 1000 字节的声音缓冲。当你为了写入数据而锁住缓冲时，你将得到两个指向内存段的返回指针。第一个可能是 900 字节，第二个则是 100 字节。关键在于你必须把头 900 字节的数据写到第一块内存部分，剩下的写到第二块内存部分。参考一下图 10.11。



注意：ptrs 为降序

图 10.11 锁定声音缓冲

下面是锁定 1000 字节声音缓冲的例子：

```

UCHAR *audio_ptr_1,          //used to retrieve buffer memory
        *audio_ptr_2;
int audio_length_1,          //length of each buffer section
    audio_length_2;
//lock the buffer
if (FAILED(lpdsbuffer->Lock(0, 1000,
    (void **)&audio_ptr_1, audio_length_1,
    (void **)&audio_ptr_2, audio_length_2,
    DSBLOCK_ENTIREBUFFER)))
    { /* error */

```

一旦你锁住了缓冲区，你就可以自由地向其中写入数据。数据可以来自一个声音文件或计算算法。当你使用完了声音缓冲后，你必须用 `Unlock()` 函数将它解开。`Unlock()` 使用两个指针和长度：

```

if (FAILED(lpdsbuffer->Unlock(audio_ptr_1, audio_length_1,
    audio_ptr_2, audio_length_2)))
    { /* problem unlocking */

```

通常，你使用完了声音缓冲后，必须像下面这样用 `Release()` 把资源释放掉：

```
lpdsbuffer->Release();
```

然而，不要破坏声音，直到你不再需要为止。否则你必须重新加载。

现在让我们看看如何使用 `DirectSound` 播放声音。

播放声音

一旦你创建了你的声音缓冲你就可以准备摇滚一下了(当然，只要你愿意，你可以随时创建和销毁声音)。`DirectSound` 有许多播放声音和改变播放参数的功能，你可以改变音量、频率、立体声等。

播放声音

要播放缓冲区中的声音数据可以使用 `Play()` 函数，其原型是：

```

HRESULT Play(
    DWORD dwReserved1, DWORD dwReserved2,    //both 0
    DWORD dwFlags);                          // control flags to play

```

惟一定义的标志就是 `DSBLPAY_LOOPING`。设定这个值可以循环播放声音。如果你只打算播放一次，把 `dwFlags` 设为 0。下面是一个循环播放的实例：

```
if (FAILED(lpdsbuffer->Play(0, 0, DSBPLAY_LOOPING)))
    { /*error*/ }
```

循环播放用于你想要重复音乐或其他素材的地方。

停止播放

一旦你启动一个声音后，在它播放完之前你很可能想终止它。这时使用 `Stop()` 函数，其原型为：

```
HRESULT Stop();    //that's easy enough
```

下面是如何停止上面启动的声音播放：

```
If(Failed(lpdsbuffer->Stop()))
    { /*error*/ }
```

现在你对 `DirectSound` 的演示有了完全的了解，测试一下 CD 上的 `DEM10_2.CPP` EXE 程序。它创建了一个 `DirectSound` 对象和一个辅助声音缓冲，然后用合成正弦波加载缓冲，最后播放。该程序很简单，但它包含了要播放声音所要了解的全部内容。

音量控制

`DirectSound` 提供控制音量或声音振幅的功能，然而这一点并不是随心所欲的。如果你的硬件不支持音量变化，`DirectSound` 就不得不用新的振幅重新混音，这就要消耗更多的控制能量。下面是所用函数原型：

```
HRESULT SetVolume(LONG lVolume);    //attenuation in decibels
```

`SetVolume()` 与你所期待的工作方式不一样。它不是通过 `DirectSound` 来增减振幅，而是控制播放的衰减。如果发送的值为 0，这相当于 `DSBVOLUME_MAX`，声音将被无衰减播放——即最大音量。如果设定为 -10000 或 `DSBVOLUME_MIN`，那么衰减度将达到最大：-100dB，这时你听不到任何声音。

最好是创建一个函数以便你传送一个 0~100 间的值或其他的值。下面的宏定义完成了这项工作：

```
#define DSVOLUME_TO_DB(volume) ((DWORD){-30*(100- volume)})
```

这里音量是从 0~100，100 是全音，0 是静音。下面是一个 50%全音播放的例子：

```
if (FAILED(lpdsbuffer->SetVolume(DSVOLUME_TO_DB(50))))
    { /*error*/ }
```

注意



如果你想知道分贝的概念，那么我会告诉你它是一个基于贝尔的声音或能量的测定单位，是以 Alexander Graham Bell 的名字命名。在电子学中许多量都是对数化测量的。分贝就是其中的一个例子。换句话说，0dB 代表没有衰减，-1dB 代表衰减到 1/10，-2dB 代表衰减到 1/100，因此-100dB 的声音只能被蚂蚁听到！记住一些范围内 dB 也是以 10 为比例系数的，所以-10dB 就是 1/10，-20dB 就是 1/100。

用频率修饰

你能用于声音控制的最酷的就是改变播放频率。这样可以改变声音的音调，你可以使声音变尖、变快、变慢或变得更悦耳。你可以使你的声音听起来像花栗鼠或 Darth Vader，如下：

```
HRESULT SetFrequency(
    DWORD dwFrequency); //new frequency from 100-100000Hz
```

下面是怎样使声音播放加快的例子：

```
if (FAILED(lpdsbuffer->SetFrequency(22050)))
{ /* error */ }
```

如果原音是以 11025Hz 采样，那么播放声音的速度就是原音的两倍，并且音调也可提高两倍，但时间却只用一半。需要吗？听完之后要删掉它。

环绕立体声

下面一个对声音有杀伤力的武器就是改变声音的立体声效或说话者的能量。例如，如果你以同样的音量在两个说话者间播放一个声音，那么感觉上声音在你的面前。但你把声音转到右边的说话者时，声音好像在向右移动。这就是所谓的动态音效，可以帮你创建 3D 音效(以粗略的方式)。

设定环绕立体声效的函数是 **Setpan()**，下面是其原型：

```
HRESULT SetPan(LONG lPan); //the pan value -10000 to 10000
```

动感值再一次使用了对数算法：0 代表位置完全处于中间，-10000 是右声道有-100dB 的衰减，10000 是左声道有-100dB 的衰减。这样做有些笨吗？但无论如何，下面是右声道衰减-5dB 的例子：

```
if (FAILED(lpdsbuffer->SetPan(-500)))
{ /* error */ }
```


用 DirectSound 反馈信息

你可能想知道有什么方法检查 DirectSound 处理声音系统的信息或正在播放的声音，比如声音是否播放完毕。当然有了。DirectSound 有许多函数做这项工作。下面是一个依赖于硬件的 DirectSound 通用函数：

```
HRESULT GetCaps(LPDSCAPS lpDSCaps); //ptr to DSCAPS structure
```

该函数简单的调用了·一个指向 DSCAPS 结构体的指针。下面是 DSCAPS 的结构供你参考（你必须参考 DirectX SDK 以得到更多的描述，但字段中的大多数都可按其字面意思理解）：

```
typedef(
DWORD   dwSize;
DWORD   dwFlags;
WORD    dwMinSecondarySampleRate;
DWORD   dwPrimaryBuffers;
DWORD   dwMaxHwMixingAllBuffers;
DWORD   dwMaxHwMixingStaticBuffers;
DWORD   dwMaxHwMixingStreamingBuffers;
DWORD   dwFreeHwMixingAllBuffers;
DWORD   dwFreeHwMixingStaticBuffers;
DWORD   dwFreeHwMixingStreamingBuffers;
DWORD   dwMaxHw3DAllBuffers;
DWORD   dwMaxHw3DStaticBuffers;
DWORD   dwMaxHw3DStreamingBuffers;
DWORD   dwFreeHw3DAllBuffers;
DWORD   dwFreeHw3DStaticBuffers;
DWORD   dwFreeHw3DStreamingBuffers;
DWORD   dwTotalHwMemBytes;
DWORD   dwFreeHwMemBytes;
DWORD   dwMaxContigFreeHwMemBytes;
DWORD   dwUnlockTransferRateHwBuffers;
DWORD   dwPlayCpuOverheadSwBuffers;
DWORD   dwReserved1;
DWORD   dwReserved2;
) DSCAPS , *LPDSCAPS;
```

你可以这样调用该函数：

```

DSCAPS dscaps ; // hold the caps
if (FAILED(lpds->GetCaps(&dscaps)))
{ /* error */ }

```

你可以检测任何想要检测的字段，并且决定你的声音硬件完成的工作。下面是一个用于 DirectSound 缓冲的类似的函数，它返回一个 DSBCAPS 结构：

```

HRESULT GetCaps(LPDSBCAPS lpDSBCaps); // ptr to DSBCAPS struct

```

这里是一个这样的 DSBCAPS 结构：

```

typedef struct{
    DWORD   dwSize;           //size of structure , you must set this
    DWORD   dwFlags;          //flags buffer has
    DWORD   dwBufferBytes;     //size of buffer
    DWORD   dwUnlockTransferRate; //sample rate
    DWORD   dwPlayCpuOverhead; //percentage of processor needed
                                // to mix this sound
} DSBCAPS, * LPDSBCAPS;

```

下面是如何检测你正在使用的声音缓冲的例子：

```

DSBCAPS dsbcaps;           //used to hold the results
//set up the struct
dsbcaps.dwSize=sizeof(DSBCAPS); //ultra important
//get the caps
if (FAILED(lpdsbuffer->GetCaps(&dsbcaps)))
{ /*error*/ }

```

这就是全部。当然，也有检索音量、立体声、频率设定的函数。但我把这部分留给你自己去看。

我们讨论的最后一个函数是决定播放状态的函数：

```

HRESULT GetStatus(LPWORD lpdwStatus); //ptr to result

```

只要用一个指向 DWORD 的指针（代表状态存储类型）从你喜欢的声音缓冲界面调用这个函数就可以了：

```

DWORD status; //used to hold status
if (FAILED(lpdsbuffer->GetStatus(&status)))
{ /*error*/ }

```

其中的状态取值如下：

- **DSBSTATUS_BUFFERLOST**——缓冲区中出现了严重的问题。
- **DSBSTATUS_LOOPING**——以循环模式播放。
- **DSBSTATUS_PLAYING**——声音正在播放，如果未设定该项值，无法播放。

读取磁盘中数据

很不幸的是 DirectSound 不支持加载声音文件。我的意思是不支持 VOC、WAV 类文件的加载，其他不存在任何问题！这是个小小的遗憾，所以你要自己编写一个加载器。问题是声音文件非常复杂，如果要把这个问题讲清楚的话，要用掉半章的篇幅。所以我给你提供一个 WAV 加载器，并且告诉你如何使用。

技巧



微软的工作人员非常辛苦地编写了他们自己的 WAV 加载器，和其他的可用函数一样，如果你喜欢你就可以直接使用。其中存在的问题是所用的 API 不是标准的并且很可能有所改变。但如果你有兴趣的话可以查一下所有的 DDUTIL*.CPP 文件，它们在安装 SDK 时的源目录之一中，通常是在实例目录下。

.WAV 文件格式

WAV 格式是电子协会 (Electronic Arts) 创建的基于 IFF 格式的 Windows 声音格式。IFF 代表对等交换文件格式 (Interchange File Format)。这是一种允许不同的文件类型通过嵌套技术用通用的标题/数据结构进行编码的标准。WAV 使用这种编码格式，尽管很清晰和有逻辑性，但读取文件却是很困难的。你必须包含很多头文件信息，其中包含很多代码，并且你必须提取声音数据。

解析代码太困难了，以至于微软创建了一套多媒体 I/O 接口 (MMIO) 来帮助加载 WAV 文件和其他相似的文件类型。库中的所有函数都有 mmio* 前缀。读者要写一个 WAV 文件并不容易，编制与游戏没有关系的程序更是叫人厌烦。所以我打算提供给你一个 WAV 文件加载器和一些解释。如果你想了解更多，可以参考一下声音文件格式方面的资料。

读取 WAV 文件

WAV 是基于块的文件格式——ID 块、格式块和数据块。通常，你需要打开一个 WAV 文件，读取头文件和格式信息，其中包含的信息有多少个频道，每个频道的位数，播放率及数据长度等等。然后加载文件中的数据。

现在，为了方便地加载和播放声音，你可以创建一个声库 API，一套全方位的多功能函数会使 DirectSound 工作起来更容易。让我们从一个记录虚拟声音的数据结构体开始，你可用它代替低水准的 DirectSound 素材：

```
//this holds a single sound
typedef struct pcm_sound_typ
{
    LPDIRECTSOUNDBUFFER dsbuffer;    //the ds buffer containing the sound
```

```

Int  state;                //state of the sound
Int  rate ;               //playback rate
Int  size;               //size of sound
Int  id;                 //id number of the sound
pcm_sound, *pcm_sound_ptr;

```

该结构体连同声音信息的重要拷贝一起很好地容纳了 DirectSound 的声音缓冲。现在让我们创建一个数组来记录系统中的声音数据:

```
pcm_sound sound_fx[MAX_SOUNDS];    //the array of secondary sound buffers
```

所以当你加载声音时就相当于打开一块空间, 建立一个 `pcm_sound` 结构。下面是 `Dsound_Load_WAV()` 函数具体如何工作:

```

int Dsound_Load_WAV(char * filename, int control_flags=DSECAPS_CTRLDEFAULT)
{
    //this function loads a .wav file ,sets up the directsound
    //buffer and loads the data into memory ,the function returns
    // the id number of the sound
    HMMIO  hwav;    //handle to wave file
    MMCKINFO  parent; //parent chunk
    Child; //child chunk
    WAVEDFORMATEX  wfmtx; //wave format structure
    Int sound_id=-1, //id of sound to be loaded
    Index; //looping variable
    UCHAR* snd_buffer, //temporary sound buffer to hold voc data
    *audio_ptr_1=NULL, //data ptr to first write buffer
    *audio_ptr_2=NULL, //data ptr to second write buffer
    DWORD audio_length_1=0, //length of first write buffer
           audio_length_2=0, //length of second write buffer
    //step one:are there any open id's ?
    for (index=0; index<MAX_SOUNDS;index++)
    {
        //make sure this sound is unused
        if (sound_fx[index].state!=SOUND_NULL)
        {
            sound_id=index;
            break;
        }
    } //end if
    // end for index
    // did we get a free ID?
    If (sound_id!=-1)
        Return(-1);
    //set up chunk info structure
    parent.ckid      =(FOURCC)0;
    parent.cksize     0;
    parent.fccType    =(FOURCC)0;
    parent.dwDataoffset=0;
}

```

```

parent.dwFlags      =0
//copy data
child parent;
//open the WAV file
if((hwav=mmioOpen(filename,NULL,MMIO_READ|MMIO_ALLOCBUF))==NULL)
    return(-1);
//descend into the RIFF
parent.fccType=mmioFOURCC('W','A','V','E');
if (mmioDescend(hwav, &parent, NULL,MMIO_FINDRIFF))
{
    // close the file
    mmioClose(hwav, 0);

    // return error, no wave section
    return(-1);
} // end if
// descend to the WAVEfmt
child.Ckid = mmioFOURCC('f','m','t',' ');
if (mmioDecend(hwav,&child, &parent,0))
{
    //close the file
    mmioClose(hwav,0));
    //return error, no format section ;
    return(-1);
}/// end if
//now read the wave format information form file
if (mmioRead(hwav, (char *)&wfmtx, sizeof(wfmtx)) !=sizeof(wfmtx))
{
    //close file
    mmioClose(hwav,0);
    //return error ,no wave format data
    return(-1);
}/// end if
// make sure that the data format is PCM
if( wfmtx.wFormatTag!=WAVE_FORMAT_PCM)
{
    //close the file
    mmioClose(hwav,0);
    //return error ,not the right data format
    return(-1);
}/// end if
//now ascend up one level, so we can access data chunk
if(mmioAscend(hwav,&child,0))
{
    //close file
    mmioClose(hwav,0);
    //return error, couldn't ascend
    return(-1);
}/// end if

```

```

//descend to the data chunk
child.ckid=mmioFOURCC('d','a','t','a');
if(mmioAscend(hwav,&child,&parent,MMIO_FINDCHUNK))
{
    //close file
    mmioClose(hwav,0);
    return error , no data
    return(-1);
}
} // end if
// finally!!! Now all we have to do is read the data in and
// set up the directsound buffer
//allocate the memory to load sound data
snd_buffer=(UCHAR *)snd_buffer,child.cksize;
//close the file
mmioClose(hwav,0);
// set rate and size in data structure
sound_fx[sound_id].rate =wfmtx.nSamplesPerSec;
sound_fx[sound_id].size =child.cksize;
sound_fx[sound_id].state =SOUND_LOADED;
//set up the format data structure
memset(&pcmwf,0,sizeof(WAVEFORMATEX));
pcmwf.wFormatTag =WAVE_FORMAT_PCM; //pulse code modulation
pcmwf.nChannels =1; //mono
pcmwf.nSamplesPerSec =11025; //always this rate
pcmwf.nBlockAlign =1;
pcmwf.nAvgBytesPerSec= pcmwf.nSamplesPerSec *pcmwf.nBlockAlign;
pcmwf.wBitsPerSample =8;
pcmwf.cbSize =0;
//prepare to create sounds buffer
dsbd.dwSize =sizeof(DSBUFFERDESC);
dsbd.dwFlags =control_flags |DSBCAPS_STATIC|
DSBCAPS_LOCSOFTWARE;
Dsbd.dwBufferBytes =child.cksize;
Dsbd.lpwfxFormat =&pcmwf;
//create the sound buffer
if(lpds->CreateSoundBuffer(&dsbd,
&sound_fx[sound_id].dsbuffer,NULL)!=DS_OK)
{
    //release memory
    free (snd_buffer);
    //return error
    return(-1);
}
} //end if
//copy data into sound buffer
if(sound_fx[sound_id].dsbuffer->lock(0,
child.cksize,
(void **)& audio_ptr_1,
&audio_length_1,
(void **)&audio_ptr_1,

```

```

&audio_length_2,)
DSBLOCK_FORMWRITECUSOR)!=DS_OK)
Return (0);
//copy first section of curcular buffer
memcpy(audio_ptr_1,snd_buffer, audio_length_1);
//copy last section of curcular buffer
memcpy(audio_ptr_2,(snd_buffer+audio_length_1), audio_length_2);
//unlock the buffer
if (sound_fx[sound_id].dsbuffer->Unlock(audio_ptr_1,
        audio_length_1,
        audio_ptr_2,
        audio_length_2)!=DS_OK)

return(0);
//release the temp buffer
free (sound_id);
)//end Dsound_Load_WAV

```

你只是简单地把文件名和标准 DirectSound 控制标志传递给函数，例如 DSBCAPS_CTRLDEFAULT 或其他。之后下面的事情发生了：

1. 函数从盘中打开了.WAV 文件，解析出其中的重要信息。
2. 函数继续创建一个 DirectSound 缓冲并且用数据填充其中。
3. 函数把信息存储在开放数组 sound_fx[]中同时返回索引值，该值作为声音的标识符。
4. 最后，你的 API 将使用标识符号来查阅声音，然后你就可以用它做你想做的事情，例如播放。

下面是一个实例：

```

//load the sound
int id=Dsound_Load_WAV("test.wav");
//manually play the sound buffer, later we will wrapper this
sound_fx[id].lpdsbuffer->Play(0, 0, DSBPLAY_LOOPING);

```

记住检验盘上的程序 DEMO10_3.CPP（不要忘了连接 DSOUND.LIB 和 WINMM.LIB）。这是 DirectSound 和 Dsound_Load_WAV()函数的完整演示。除次之外，程序可通过滚动条实时地控制声音，所以它不仅很酷，而且你可以看到如何对你的应用增加滚动条！

当然，我打算使用所有的 DirectSound 素材，给你介绍全部的库（T3DLIB3.CPPH），但首先还是让我们看一下 DirectMusic。

DirectMusic：伟大的试验

DirectMusic 是 DirectX 中最令人激动的一个组件。正如我以前所说，编写数字声音软件很困难，但编制播放 MIDI 文件的软件却很轻松！DirectMusic 可以播放 MIDI 文件，但

其功能远不止这一点。下面列出了更多：

- 支持 DLS (downloadable sounds) 乐器。这就意味着在用 DirectMusic 播放 MIDI 文件时，不管你的硬件是什么类型，其效果都一样。
- 支持实时乐曲合成，DirectMusic 允许为你的歌曲建立模板、个人存档和各种气氛。然后 DirectMusic 会提取你的歌曲数据，并且实时地重新编写乐曲及产生新的音乐。
- 理论上支持无限数据量的 MIDI 频道，但实际上局限于你的 PC 机的控制能量。通常 MIDI 支持 16 个频道或 16 个单声道。在 DirectMusic 下有 65536 个频道组，因此你几乎可以无限制的同时播放任意多首乐曲。
- 如果可能的话可以使用硬件加速，但微软的软件合成器是默认的设备，并且其效果就像波导或波表合成的一样。

关于 DirectMusic 的惟一不足就是它像 Direct3D 一样复杂！我曾看过有关它的在线文档（约 500 页），我可以告诉你在我的脑子里根本就没有简单的概念，但它确实功能强大。幸运的是你只是要播放一个补丁 MIDI 文件，所以我会引导你怎样做，怎样创建关于 DirectMusic 的 API，以使你可以加载和播放 MIDI 文件。

DirectMusic 的结构

DirectMusic 相当庞大，所以我不打算涉及它的任何细节。它可以构成一本完整书籍的主题。不过我要讨论一下你将使用的界面。看一下图 10.12，那是 DirectMusic 的几个主要界面。

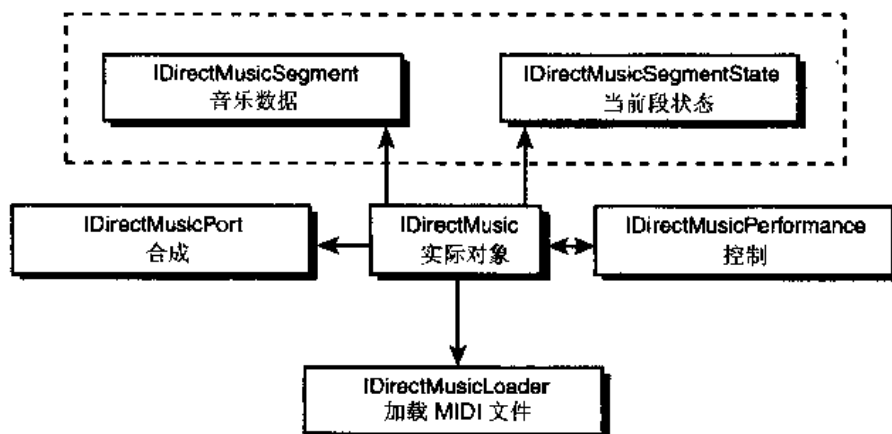


图 10.12 DirectMusic 的主界面

这些界面的说明如下：

IDirectMusic——这是 DirectMusic 的主界面，但与 DirectDraw 和 DirectSound 不同，你

并不需要使用 DirectMusic。当你创建一个 DirectMusic 对象时，它是默认和隐含创建的。

IDirectMusicPerformance——这是你关心的主界面。建立的对象对播放的音乐数据进行控制。建立该画面的同时也创建了一个 **IDirectMusic** 对象。

IDirectMusicLoader——它用于加载所有的数据，包括 MIDI、DLS 等。你可以用它来加载盘上的 MIDI 文件，所以你有了一个 MIDI 加载器——很简单吧！

IDirectMusicSegment——这代表一个音乐数据块；每个你加载的 MIDI 文件都通过该界面表示出来。

IDirectMusicSegmentState——这是与段相关的。但它是与当前段的状态有关，而不是与数据有关。

IDirectMusicPort——这是你的 MIDI 音乐输出的位置。大多数情况下它是微软合成器，但你可以始终设定其他硬件加速的端口作为输出位置。

一般说来，DirectMusic 是一个有 DSP (Digital Signal Processing) 功能的 MIDI 到数字的实时转换器。在讨论 DirectSound 时我就提到过 MIDI，它存在的问题就是在不同的硬件上播放出不同的 MIDI 音效。DirectMusic 在 DLS 文件中通过使用乐器的单音采样来处理这个问题。所以只要你制作声乐，就可以使用默认的 DLS 文件或创建你自己的乐器文件。乐器以自然状态数字化；同你的音乐一样，数字声音通过 D/A 转换播放起来都是一样的，因此音乐听起来始终是相同的，参考一下图 10.13 就明白了。

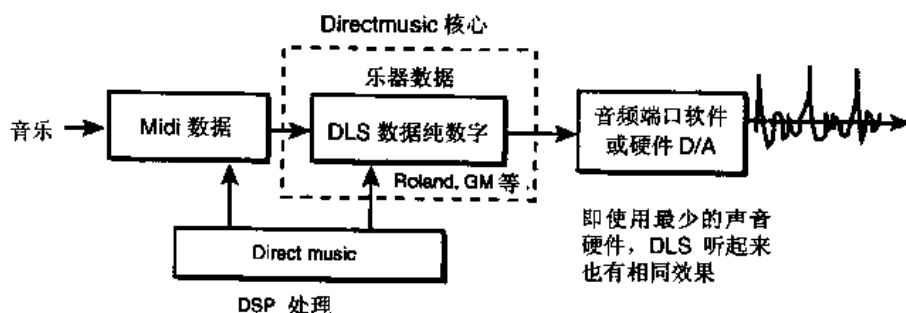


图 10.13 DirectMusic 依赖于数字采样而非合成

注意

可以在任何一部机子上装载的默认的 DLS 乐器是 RolandGM/GS (GeneralMIDI)。它们的效果非常好，但你不可以任何方式改变它们——Roland 并不想让你把它们搞坏。

我知道，任何东西都好像比它需要的要复杂。但复杂是有好处的，因为它可推动技术的进步和革新，这就是 DirectMusic 的基础。

启动 DirectMusic

DirectMusic 是 DirectX 中第一个完全 COM 化的组件，这意味着在输入库中没有任何函数可帮助你创建 COM 对象。当然你自己必须调用 COM 库来创建 COM 对象。因此每个应用都需要的文件就是 DirectMusic 头文件，没有任何输入库文件。所用的头文件有：

```
Dmksctrl.h
Dmusicl.h
Dmusicf.h
```

只要在你的应用中包含这些文件，COM 就会完成其余的工作，让我们看一下完整的情况。

初始化 COM

首先你必须调用 CoInitialize() 初始化 COM：

```
// initialize com
if(FAILED(CoInitialize(NULL)))
{
    //Terminate the application
    return (0);
} // end if
```

这些代码应放在你的应用的开头，在直接调用 COM 之前。如果你有其他的 COM 调用并且已经执行了，那就不必再考虑这一点了。

创建表演

下一步是创建总界面，它是 DirectMusic 的演奏界面。这个界面的创建也要创建一个内部 IDirectMusic 界面，但你不需要它，因此将它消去。创建界面基于纯 COM，使用有界面标识符和类标识符的 CoCreateInstance() 函数，另外还要存储新界面指针。看一下下面的调用：

```
// the directmusic performance manager
if (FAILED(CoCreateInstance(CLSID_DirectMusicPerformance,
                            NULL,
                            CLSCTX_INPROC,
                            IID_IDirectMusicPerformance,
                            (void**) &dm_perf)))
{
    //return null
    return(0);
}
```

```
}// end if
```

看起来有些神秘，其实很有逻辑性。完成之后，dm_perf 就可以使用了，也就可以调用界面函数了。你需要调用的是初始化函数 `IDirectMusicPerformance::Init()`。下面是它的原型：

```
HRESULT Init(IDirectMusic ** ppDirectMusic,
             LPDIRECTSOUND pDirectSound,
             HWND hWnd);
```

ppDirectMusic 是 IDirectMusic 界面的地址。如果你没有创建则设定为 NULL。
PDirectSound 是一个指向 IDirectSound 的指针。

警告



这是很重要的，所以要仔细阅读；如果你想同时使用 DirectSound 和 DirectMusic，你必须首先启动 DirectSound，然后在 Init() 中启动 DirecSound 对象。然而，如果你正在单独使用 DirectMusic，传递 NULL，DirectMusic 就会创建一个 DirectSound 对象。这是很必要的，因为 DirectMusic 最终要经过 DirectSound，如图 10.14 所示。

你必须传递主 DirectSound 对象指针或者不使用时传递 NULL。如果用于演示的话，用后者就可以。最后，你必须向窗口传递句柄。这很容易作到，以下是代码：

```
// initialize the performance, check if directsound is on-line if so, use
// the directsound object, otherwise create a new one
if(FAILED(dm_perf->Init(NULL, NULL, main_window_handle)))
{
return(0) ; // Failure -performance not initialized
} // end if
```

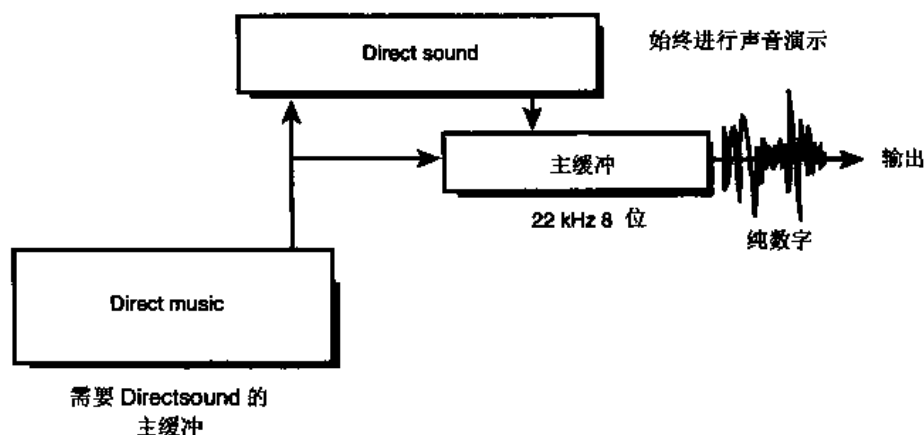


图 10.14 DirectMusic 与 DirectSound 的关系

增加播放端口

下一步是运行 `DirectMusic`，为数字数据输出创建一个端口。如果你愿意，你可以通过列举法查阅所有的有效端口，或者就用微软的合成器作为默认设备，那是我惯用的风格——很简单。要增加一个演奏端口，使用 `IDirectMusicPerformance::AddPort()`，其原型为：

```
HRESULT AddPort(IDirectMusicPort * pPort);
```

这里 `pPort` 是一个指向前面创建的用于播放的端口指针。然而，只要用 `NULL` 值和默认的软件合成器就可用了：

```
// add the port to the performance
if(FAILED(dm_perf->AddPort(NULL)))
{
    return (0); // Failure-port not initialized
} // end if
```

加载 MIDI 段

下一步是创建一个 `IDirectMusicLoader` 对象，以便可以加载你的 `MIDI` 文件。这再一次用到了低级 `COM` 调用来完成，但效果并不差。

创建加载器

下面的代码是用于创建加载器的：

```
// the directmusic loader
IDirectMusicLoader* dm_loader=NULL;
//create the loader to load object(s) such as midi file
if(FAILED(CoCreateInstance(
    CLSID_DirectMusicLoader,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicLoader,
    (void**)&dm_loader)))
{
    //error
    return(0);
} // end if
```

很有意思吧，许多界面都是在内部创建——包括一个 IDirectMusic 对象和一个 IDirectMusicPort 对象——你可能并不知道。大多数情况下你根本就不需要调用这些界面的函数，所以很酷吧。

加载 MIDI 文件

要加载 MIDI 文件，你必须告诉加载器去哪里加载及加载什么，然后告诉它创建一个段把文件放进去。我已经创建了一个函数和数据结构体来完成这项工作，因此现在就可以介绍给你。首先，记录 MIDI 音乐段（DirectMusic 习惯称为块）的数据结构体叫做 DMUSIC_MIDI，如下所示：

```
typedef struct DMUSIC_MIDI_TYP
{
    IDirectMusicSegment      *dm_segment;    //the directmusic segment
    IDirectMusicSegmentState *dm_segment;    //the state of the segment
    Int                      id;             //the id of this segment
    Int                      state;          state of mild song
}DMUSIC_MIDI, *DMUSIC_MIDI_PTR;
```

它被用于记录每一个 MIDI 段。但你可以在游戏中加载好几首曲子，所以让我们定义一个数组：

```
DMUSIC_MIDI dm_midi[DM_NUM_SEGMENTS];
```

下面是 DM_NUM_SEGMENTS 的定义：

```
#define DM_NUM_SEGMENTS 64 //number of midi segments that can be cached in
                           //memory
```

OK，记住了吧！看一下下面的 DMusic_Load_MIDI()函数。它是很重要的，所以要花一些时间，同时注意 DirectMusic 函数使用的比较有趣的串：

```
int Dmusic_Load_MIDI(char *filename)
{
    // this function loads a midi segment
    DMUS_OBJECTDESC ObjDesc;
    HRESULT hr;
    IDirectMusicSegment * pSegment = NULL;
    Int index; // loop var
    // look for open slot for midi segment
    int id = -1;
    for (index = 0; index < DM_NUM_SEGMENTS; index++)
    {
        // is this one open
        if (dm_midi[index].state == MIDI_NULL)
```

```

    //validate id, but don't validate object until loaded
    id = index;
    break;
} // end if
} // end for index
// found good id?
if (id != -1)
    Return( 1);
// get current working directory
char szDir[_MAX_PATH];
WCHAR wszDir[_MAX_PATH];
If(!_getcwd( szDir, _MAX_PATH ) == NULL)
{
    return (-1);
} // end if
MULTI_TO_WIDE(wszDir, szDir);
// tell the loader where to look for files
hr = dm_loader->SetSearchDirectory(GUID_DirectMusicAllTypes,
                                   wszDir, FALSE);

if (FAILED(hr))
{
    return (-1);
} // end if
// convert filename to wide string
WCHAR wfilename[_MAX_PATH];
MULTI_TO_WIDE(wfilename, filename);
// setup object description
DD_INIT_STRUCT(objDesc);
ObjDesc.guidClass = CLSID_DirectMusicSegment;
wcscpy (objDesc, wszFilename, wfilename );
objDesc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME;
//load the object and query it for the IDirectMusicSegment interface
//This is done in a single call to IDirectMusicloader::GetObject
//note that loading the object also initializes the tracks and does
//everything else necessary to get the MIDI data ready for playback.
hr = dm_loader->GetObject(&objDesc, IID_IDirectMusicSegment,
                         (void**) &pSegment);

if (FAILED(hr))
    return(-1);
// ensure that the segment plays as a standard MIDI file
// you now need to set a parameter on the band track
//Use the IDirectMusicSegment::SetParam method and let
// DirectMusic find the trackby passing -1
// (or 0xFFFFFFFF) in the dwGroupBits method parameter.
Hr= pSegment ->Setparam(GUID_StandardMIDIFile, -1, 0, 0, (void*)dm_perf);
if (FAILED(hr))
    return (-1);
// This step is necessary because DirectMusic handles program changes and
// bank selects differently for standard MIDI files than it does for MIDI.

```

```

// content authored specifically for DirectMusic.
// The GUID_StandardMIDIFile parameter must.
// be set before the instruments are downloaded.
// The next step is to download the instruments.
// This is necessary even for playing a simple MIDI file
// because the default software synthesizer needs the DLS data
// for the General MIDI instrument set
// If you skip this step, the MIDI file will play silently.
// Again, you call Setparam on the segment,
// this time specifying the GUID_Download parameter:
hr= pSegment->SetParam(GUID_Download, -1, 0, 0, (void*)dm_perf));
if (FAILED(hr))
    return(-1);
// at this point we have MIDI loaded and a valid object
DirectX and 2D Fundamentals
    dm_midi[id].dm_segment = pSegment;
    dm_midi[id].dm_segment = NULL;
    dm_midi[id].state      = MIDI_LOADED;
    //return id
    return(id);

} // end Dmusic_Load_MIDI

```

这个函数并不复杂。它首先寻找一个开放的数组 `dm_midi[]` 来加载新的 MIDI 段、设定搜索路径、创建段、加载段然后释放。函数提取文件名后返回一个容纳数据结构体段的数组索引标识符。

操作 MIDI 段

许多界面函数（方法）都用于 `IDirectMusicSegment` 界面，它代表一个加载了的 MIDI 段。如果你有兴趣，可以在 SDK 中查阅一下。但两个看起来最重要的功能就是播放和停止，对吧？不是吗，它们是 `IDirectMusicPerformance` 界面的组成部分，而不是 `IDirectMusicSegment` 界面的组成部分。这一点的意义在于你的思考：表演对象就像表演的指挥者，每件事情都要经过他。像我的女朋友所说“无论什么”。

播放 MIDI 段

假定你已经用 `Dmusic_Load_MIDI()` 或人工加载了一段，让 `dm_segment` 作为指向段的界面指针。那么用演奏对象播放它就用 `IDirectMusicPerformance::PlaySegment()` 函数，原型如下：

```
HRESULT PlaySegment(
```

```

IDirectMusicSegment * pSegment,    //segment to play
DWORD dwFlags,                    // control flags
__int64 i64StartTime,              //when to play
IDirectMusicSegmentState ** ppSegmentState); //state holder

```

通常设定控制标志和初始时间为 0。惟一叫人担心的参数就是段和段状态。这里是一个播放 `dm_segment` 及把状态存储在 `dm_segstate` 中的例子：

```
dm_perf->PlaySegment(dm_segment, 0, 0, &dm_segstate);
```

其中 `dm_segstate` 是 `IDirectMusicSegmentState` 类型，被用于跟踪段的播放。每一个数组组元 `dm_midi[]` 都有一个拷贝，但如果你自己完成，别忘了给自己传送一份。

停止 MIDI 段

要在播放时停止一个段，使用 `IDirectMusicPerformance::Stop()` 函数，原型如下：

```

HRESULT Stop(
IDirectMusicSegment * pSegment,    //segment to stop
IDirectMusicSegmentState *pSegmenState,          //state
MUSIC_TIME mtTime,                  //when to stop
DWORD dwFlags); //control flags

```

与 `Play()` 类似，大多数参数你都不用考虑，但段本身却不能忽略。下面是一个停止 `dm_Segment` 的例子：

```
dm_perf->Stop(dm_segment, NULL, 0, 0);
```

如果你想停止所有正在播放的段，则将 `dm_segment` 设为 `NULL`。

检测 MIDI 段的状态

我们常常想知道一首曲子是否已播放完。这时使用 `IDirectMusicPerformance::IsPlaying()` 函数。它很简单地检测播放的段，如果仍然在播放就返回 `S_OK`。这里是一个例子：

```

if(dm_perf->IsPlaying(dm_segment, NULL)==S_OK)
    { /* still playing */ }
else
    { /* not playing */ }

```

释放 MIDI 段

当你完成了段的播放后，必须释放资源。第一步是调用 `IDirectMusicSegment::SetParam()` 卸载 DLS 乐器数据，然后用 `Release()` 释放界面指针。下面是代码：


```
//unload the instrument data
dm_segment->SetParam(GUID_Unload, -1, 0, 0, (void *) dm_perf);
//Release the segment and set to null
dm_segment->Release();
dm_segment = NULL; //for good measure
```

关闭 DirectMusic

当你使用完了 DirectMusic 后，必须关闭和释放对象资源，释放加载器和其他所有的段（参考下面的代码）。最后你必须关闭 COM，除非其他地方正在使用。下面是控制的例子：

```
// If there is any music playing , stop it. This is
// not really necessary, because the music will stop when
// the instruments are unloaded or the performance is
//closed down
if(dm_perf)
    dm_perf->Stop(NULL, 0, 0);
//***delete all the midis if they already haven't been
//CloseDown and Release the performance object.
If(dm_perf)
{
    dm_perf->CloseDown();
    dm_perf->Release();
}
//Release COM
CoUninitialize();
```

DirectMusic 实例

我已经在 CD 上创建了程序 DEMO10_4.CPPIEXE，这个例子没有用 DirectSound 和 DirectX 的其他组件，而是使用 DirectMusic。它只是简单地加载了一个 MIDI 文件然后进行播放。可以参考和测试一下。完成之后回来再看看用最新安装的库文件的部分 T3DLIB3.CPPIH 来完成这一切是多么容易。

T3DLIB3 声音和音乐库

我已经介绍了所有的声音和音乐技术，那是我们曾经编译过的。它们用于创建游戏工程中的下一个组件，T3DLIB3。它由两个主要的资源文件组成：

- T3DLIB3.CPP——主 C/C++ 资源

- T3DLIB3.H——头文件

你同样需要包含进 DirectSound 输入库 DSOUND.LIB，从而使各模块相互连接起来。然而，因为 DirectMusic 是纯 COM，它并没有什么输入库，更没有 DMUSIC.LIB。换句话说，你仍然需要把你的编译器指向 DirectSound 和 DirectMusic 的头文件，以便在编译时找到它们。再一次提醒你，它们是：

```
DSOUND.H
DMKSCTRL.H
DMUSICI.H
DMUSICC.H
SMUSICF.H
```

把这些文件记住后，让我们看一下 T3DLIB3.H 头文件的主要元素。

头文件

头文件 T3DLIB3.H 包含类型、宏和 T3DLIB3.CPP 的外观。下面是可以在头文件中找到的定义：

```
//number of midi segments that can be caught in memory
#define DM_NUM_SEGMENTS 64
//midi object state define
#define MIDI_NULL 0 //this midi object is not loaded
#define MIDI_LOADED 1 //this midi object is loaded
#define MIDI_PLAYING 2 //this midi object is loaded and playing
#define MIDI_STOPPED 3 //this midi object is loaded, but stopped
#define MAX_SOUNDS 256 //max number of sounds in system at once
//digital sound object state defines
#define SOUND_NULL 0// ''
#define SOUND_LOADED 1
#define SOUND_PLAYING 2
#define SOUND_STOPPED 3
```

没有太多的宏；就一个帮助转换 0~100 到微软分贝范围内的宏和一个转换多字节特征的宏：

```
#define DSVOLUM_TO_DB(volume) ((DWORD)(-30*(100-volume)))
//convert from multibyte format Unicode using the following macro
#define MULTI_TO_WIDE(x,y) MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED,
y, 1, x, _MAX_PATH)
```

警告



该书的宽度太小而不能把宏定义放在一行。但实际应用时必须在一行。

下面是声音引擎的类型。

类型

首先是 DirectSound 对象。有两种类型的声音引擎：一种记录数字采样，另一种记录 MIDI 段。

```
//this holds a single sound
typedef struct pcm_sound_typ
{
    LPDIRECTSOUNDBUFFER dsbuffer; // the directsound buffer
    // containing the sound
    int state; //state of the sound
    int rate ; //playback rate
    int size ; //size of sound
    int id; // id number of the sound
}pcm_sound, *pcm_sound_ptr;
```

下面是 DirectMusic 段的类型：

```
//directmusic MIDI segment
typedef struct DMUSIC_MIDI_TYP
{
    IDirectMusicSegment *dm_segment; //the directmusic segment
    IDirectMusicSegmentState *dm_segstate; // the state of the segment
    Int id; // the id of this segment
    Int state //state of midi song
}DMUSIC_MIDI, *DMUSIC_MIDI_PTR;
```

声音和 MIDI 段都将被引擎存储在下面的两个结构中。现在让我们看一下全局量。

全局控制

T3DLIB3 包含许多全局量。首先是为 DirectSound 系统定义的全局量：

```
LPDIRECTSOUND lpds; //directsound interface pointer
DSBUFFERDESC dsbd; //directsound description
DSCAPS dscaps; //directsound caps
HRESULT dsresult; //general directsound result
DSBCAPS dsbcaps; //directsound buffer caps
```

```
Pcm_sound sound_fx[MAX_SOUNDS]; //array of sound buffers
WAVEFORMATEX *pcmwf; //generic waveformat structure
```

以下是为 DirectMusic 定义的全局量:

```
//direct music globals
//the directmusic performance manger
IDirectMusicPerformance *dm_perf;
IdirectMusicLoader *dm_loader; //the directmusic loader
//this hod all the directmusic midi objects
DMUSIC_MIDI dm_midi[DM_NUM_DEGMENTS];
Int dm_active_id; //currently active midi segment
```

说 明



黑体部分是记录声音和 MIDI 段的数组。

除了直接进入界面之外你不应该混乱地使用这些量。通常, API 会为你完成所有的工作, 但这些全局量仍存在, 即使你想支解它们。

库有两部分: DirectSound 和 DirectMusic。让我们先看看 DirectSound, 然后再看 DirectMusic。

DirectSound API 封装

DirectSound 可以很简单, 也可很复杂, 主要取决于你怎样使用。如果你想作为一个完全 API, 你就要打算停止使用 DirectSound 的大多数函数本身。但如果你想作为一个可初始化 DirectSound 和加载及播放特殊格式声音的简单 API 的话, 那是非常容易包含进几个函数的。

所以我所做的工作就是在本章把你的大部分工作放入 DirectSound 部分, 并且为你介绍正式的函数。除此之外, 我创建了一个围绕声音系统的提取法, 所以你可以用在加载过程中通过你的标识符(与 DirectMusic)来查阅声音。这样, 你可用标识符来播放声音、检测状态或终止播放。用这种方法就避免了使用许多你容易搞混的界面指针。新的 API 支持如下功能:

- 单独调用就可初始化和关闭 DirectSound。
- 用 11kHz, 8-bit 加载 WAV 文件。
- 播放加载声音文件。
- 停止播放。

- 检测播放状态。
- 改变音量、播放率或立体声。
- 从内存中删除声音数据。

让我们一个一个地看看这些函数。

注 意



除非有其他的声明，否则只要调用成功就会返回 TRUE (1)，失败就返回 FALSE (0)。

函数原型:

```
Int Dsound_Init(void);
```

目的:

Dsound_Init()初始化 DirectSound 系统。它创建 DirectSound 的 COM 对象，设定优先级等。如果想处理声音，只要在你的应用开头调用该函数就可以。这里是一个例子：

```
If(!Dsound_Init(void)
/*error*/}
```

函数原型:

```
Int Dsound_Shutdown(void);
```

目的:

Dsound_Shutdown()关闭和释放所有的在 Dsound_Init()中初始化和创建的 COM 界面。然而，Dsound_Shutdown()并不释放分配给声音的所有内存。你必须用另一个函数解决这个问题。可以这样做：

```
If (!Dsound_Shutdown())
/*error*/}
```

函数原型:

```
Int Dsound_Load_WAV(char *filename);
```

目的:

Dsound_Load_WAV()创建了一个 DirectSound 缓冲，把声音数据文件加载到内存中，为播放做好准备。该函数从盘上搜寻完全路径和被加载的文件名（包括扩展名为.WAV 的文件）。如果执行成功则返回一个非负的标识符值。你必须保存这个值，因为它作为句柄来控制声音。如果函数找不到文件或加载的文件太多，它就会返回-1。下面是加载一个叫 FIRE.WAV 的 WAV 文件的例子：

```

int fire_id=Dsound_Load_WAV("FIRE.WAV");
//test for error
if (fire_id!=-1)
{ /*error*/ }

```

当然，怎样保存标识符值则根据你自己的爱好了。你可以用一个数组或其他的东西。

最后，你可能想知道声音数据的位置和怎样混合使用。如果你确实必须知道，你可以用 `pcm_sound` 数组 `sound_fx[]` 访问数据，用返回的标识符作为索引。例如，下面的例子就是用标识符 `sound_id` 如何访问 `DirectSound` 缓冲：

```
sound_fx[sound_id].dsbuffer
```

函数原型：

```
Int Dsound_Replicate_Sound(int source_id); // id of sound to copy
```

目的：

`Dsound_Replicate_Sound()` 被用于以不拷贝存储声音内存的方式复制声音。例如，有一种打枪的声音并且你想连发三枪，一枪接着一枪。现在的惟一途径就是复制三份枪响声到三块不同的 `DirectSound` 缓冲中，这样做很浪费内存。

当然，有一个新方法——创建缓冲的副本，这是可能的，但实际的声音数据例外。不是拷贝它，你只要用一个指针指向它，`DirectSound` 会很灵巧地把副本作为声源发出同样的声音。如果你想发出八声枪响，只需加载一次枪响数据，执行七次拷贝就可取得八声枪响的效果。复制声音与正常声音差不多。除了用 `Dsound_Load_WAV()` 加载和创建之外，你可用 `Dsound_Replicate_Sound()` 来完成。需要吗？好，我头都快晕了！下面是产生八声枪响的例子：

```

int gunshot_ids[8]; // this holds all the id's
//load in the master sound
gunshot_ids[0]=Load_WAV("GUNSHOT.WAV");
//now make copies
for (int index=1; index<8; index++)
gunshot_ids[index]=Dsound_Replicate_Sound(gunshot_ids[0]);

// use gunshot_ids[0..7] anyway you wish , they all go bang!

```

函数原型：

```

Int Dsound_Play_Sound(int id,          //id of sound to play
                      Int flags=0,     //0 or DSBPLAY_LOOPING
                      Int volume=0,    //unused
                      Int rate=0,      //unused
                      Int pan=0);      //unused

```

目的:

Dsound_Play_Sound()播放已经装载的声音。你只要连同单曲播放标志 0 或循环标志 DSBPLAY_LPPING 和声音标识符号一起传送,声音就开始播放了。如果声音已经开始播放,那么它将会从开头重新播放。下面是装载和播放声音的例子:

```
Int fire_id=Dsound_Load_WAV("FIRE.WAV");
Dsound_Play_Sound(fire_id, 0);
```

或者你干脆不要考虑 0 标志,因为它是默认参数:

```
Int fire_id=Dsound_Load_WAV("FIRE.WAV");
Dsound_Play_Sound(fire_id);
```

用两种方法都可以播放和停止 FIRE.WAV 文件一次。要循环播放,将标志参数设为 DSBPLAY_LOOP 就可以了。

函数原型:

```
Int Dsound_Stop_Sound(int id);
Int Dsound_Stop_All_Sound(void);
```

目的:

Dsound_Stop_Sound()用于停止正播放的单曲。你只要传递播放声音的标识符就可以了。

Int Dsound_Stop_All_Sound()将终止所有当前播放的声音。下面是停止 fire_id 声音的例子:

```
Dsound_Stop_All_Sound(fire_id);
```

在你的游戏结束时,从播放状态停止所有的声音当然是个好主意。你可以分开调用 Dsound_Stop_Sound()和 Dsound_Stop_All_Sound(),像下面这样:

```
//_system shutdown code
Dsound_Stop_All_Sound(void);
```

函数原型:

```
Int Dsound_Delete_Sound(int id); // id of sound to delete
Int Dsound_Delete_All_Sound(void);
```

目的:

Dsound_Delete_Sound()从内存中删除一个声音,同时释放相关的 DirectSound 缓冲。如果声音正在播放,函数首先会终止它。Dsound_Delete_All_Sound()则删除加载的所有声音。下面是一个删除 fire_id 声音的例子:

```
Dsound_Delete_Sound(fire_id);
```

函数原型:

```
Int Dsound_Status_Sound(int id);
```

目的:

Dsound_Status_Sound()根据标识符号检测加载声音的状态。你需要做的就是传送标识符给该函数，函数将返回如下的值:

- DSBSTATUS_LOOPING——声音正在以循环模式播放。
- DSBSTATUS_PLAYING——声音正在以单曲模式播放。

如果返回的不是这两个值的话说明声音没有播放。下面是一个完整的例子，声音在等待直到播放完成和被删除:

```
// initialize DirectSound
Dsound_Dsound_Init();
//load a sound
int fire_id_Sound(fire_id);
//wait until the sound is done
while(Dsound_Sound_Status(fire_id)&
      {DSBSTATUS_LOOPING|DSBSTATUS_PLAYING});
//delete the sound
Dsound_Delete_Sound(fire_id);
//shutdown DirectSound
Dsound_Dsound_Shutdown();
```

很酷吧。比 DirectSound 人工要求的上百条代码要好的多吧!

函数原型:

```
Int Dsound_Set_Sound_Volume(int id,    // id of sound
                             Int vol); // volume from 0-100
```

目的:

Dsound_Set_Sound_Volume()实时地改变音量。连同 0~100 间的值一起传送声音的标识符值，音量将立即改变。下面是音量减小一半的例子:

```
Dsound_Set_Sound_Volume(fire_id, 50);
```

你可以始终把音量定为最大:

```
Dsound_Set_Sound_Volume(fire_id, 100);
```


函数原型:

```
int Dsound_Set_Sound_Freq(
    Int id,      //sound id
    Int freq);  //new playback rate from 0-100000
```

目的:

Dsound_set_Sound_Freq() 改变声音的播放频率。因为所有的声音都以 11kHz 的频率加载, 下面是如何改变播放率的例子:

```
Dsound_Set_Sound_Freq (fire_id, 22050);
```

要使你的声音像 Darth Vader 一样, 可以这样做:

```
Dsound_Set_Sound_Freq(fire_id, 6000);
```

函数原型:

```
Int DSound_Set_Sound_Pan(
    Int id,      //sound id
    Int pan);  //panning value frome -10000 to 10000
```

目的:

DSound_Set_Sound_Pan() 调整左右声道的声音强度。-10000 是右声道最强, 10000 是左声道最强。如果你想让左右声道声强相同, 只要设定 0 值就可以了。下面是设定右声道的例子:

```
Dsound_Set_Sound_Pan(fire_id, 10000);
```

DirectMusic API

DirectMusic API 甚至比 DirectSound API 更简单。我已经创建了一个初始化 DirectMusic 及创建所有允许你加载和播放 MIDI 文件的 COM 对象的函数。下面是其基本功能:

- 初始化和关闭 DirectMusic。
- 从盘上加载 MIDI 文件。
- 播放 MIDI 文件。
- 停止正在播放的文件。
- 检测 MIDI 段的状态。
- 如果已经初始化了自动连接 DirectSound。
- 从内存中删除 MIDI 段。

让我们详细地看看每个函数。

注 意



除非有其他的声明，否则只要调用成功就会返回 TRUE (1)，失败则返回 FALSE (0)。

函数原型:

```
Int Dmusic_Init(void);
```

目的:

Dmusic_Init()初始化 DirectMusic 并创建所有需要的 COM 对象。你可以在调用其他 DirectMusic 库之前调用这个函数。除此之外，如果你想使用 DirectSound，一定要在调用 Dmusic_Init()前初始化 DirectSound。下面是使用这个函数的例子：

```
If(!Dmusic_Init())
{ /*error*/ }
```

函数原型:

```
int Dmusic_Shutdown(void);
```

目的:

Dmusic_Shutdown()关闭 DirectMusic 引擎。除了卸载所有的 MIDI 段之外，还释放所有的 COM 对象。在你的应用最后调用该函数，但一定要在关闭 DirectSound 之前（如果你有 DirectSound 支持）。下面是一个例子：

```
If (!Dmusic_Shutdown())
{ /* error*/ }
// now shut down DirectSound...
```

函数原型:

```
Int Dmusic_Load_MIDI(char * filename);
```

目的:

Dmusic_Load_MIDI()把一个 MIDI 段加载进内存，同时在 midi_ids[]数组中分配一个记录。该函数返回加载的 MIDI 段的标识符，不成功时则返回-1。返回的标识符作为其他调用的参考。下面是加载一对 MIDI 文件的例子：

```
// load files
int explode_id= Dmusic-Load_MIDI("explosion.mid");
int weapon_id= Dmusic-Load_MIDI("laser.mid");
// test files
if(explode_id== -1 || weapon_id== -1)
    { /*there was a problem */ }
```

函数原型:

```
Int Dmusic_Delete_MIDI(int id);
```

目的:

Dmusic_Delete_MIDI () 从系统中删除一个加载的 MIDI 段。只要提供标识符就可以删除了。下面是删除加载 MIDI 文件的例子:

```
If(!Dmusic_Delete_MIDI(explode_id) ||
    !Dmusic_Delete_MIDI(weapon_id))
    { /*error*/ }
```

函数原型:

```
Int Dmusic_Delete_All_MIDI(void);
```

目的:

Dmusic_Delete_All_MIDI 简单地从系统中删除所有的 MIDI 段。下面是一个例子:

```
//delete both of our segments
if(!Dmusic_Delete_All_MIDI())
    { /*error*/ }
```

函数原型:

```
Int Dmusic_Play(int id);
```

目的:

Dmusic_Play() 从开头播放 MIDI 段。只要提供要播放段的标识符就可以了。下面是一个例子:

```
// load file
int explode_id=Dmusic_Load_MIDI("explosion.mid");
```

```
// play it
if (!Dmusic_Play(explode_id))
    { /*error*/ }
```

函数原型:

```
Int Dmusic_Stop(int id);
```

目的:

Dmusic_Stop()停止当前播放的段。如果对已经停止的段进行操作,那么该函数没有反应。下面是一个例子:

```
//stop the laser blast
if(!Dmusic_Stop(weapon_id))
    { /*error*/ }
```

函数原型:

```
Int Dmusic_Status_MIDI(int id);
```

目的:

Dmusic_Status_MIDI()检测所有的 MIDI 段。状态代码有:

```
#define MIDI_NULL    0    //this midi object is not loaded
#define MIDI_LOADED  1    // this midi object is loaded
#define MIDI_PLAYING  2    //this midi object is loaded and playing
#define MIDI_STOPPED  3    // this midi object is loaded, but stopped
```

下面是改变 MIDI 段的完整的例子:

```
//main game loop
while(1)
{
    if(Dmusic_Status(explode_id)==MIDI_STOPPED)
        game_state=GAME_MUSIC_OVER;
} //end while
```

要得到使用库的演示,可参见 DEMO10_5.CPP1EXE 和 DEMO10_6.CPP1EXE 程序。第一个程序是使用新库的演示,它可以在一个主菜单内选定一个 MIDI 文件,并且立即播放。第二个程序是同时使用 DirectSound 和 DirectMusic 的混合模式的应用。第二个程序的关键点是 DirectSound 必须首先初始化。声音库会检测这一点,然后与 DirectSound 连接。否则

声音库会创建自己的 DirectSound 对象。

警告



DEMO10_5.CPP 和 DEMO10_6.CPP 中分别使用了 DEMO10_5.RC 和 DEMO10_6.RC 中包含的外部光标、图标及菜单。因此编译时要确保包括了这些文件。同时编译时也需要包括 T3DLIB3.CPPH, 不过这一点想必你已经知道了。

总结

这一章涵盖了许多内容。你了解了一点声音和音乐的性质, 怎样合成及怎样记录声音。然后了解了 DirectSound 和 DirectMusic, 创建了一个库, 看了许多演示。我想进入 DirectSound3D 部分, 而不涉及更高深的 DirectMusic, 那就看你的喜好了。现在你了解了制作游戏所需要的东西, 也是为下一章所做的准备。

第三部分

编程核心

第十一章

算法、数据结构、内存管理及多线程

第十二章

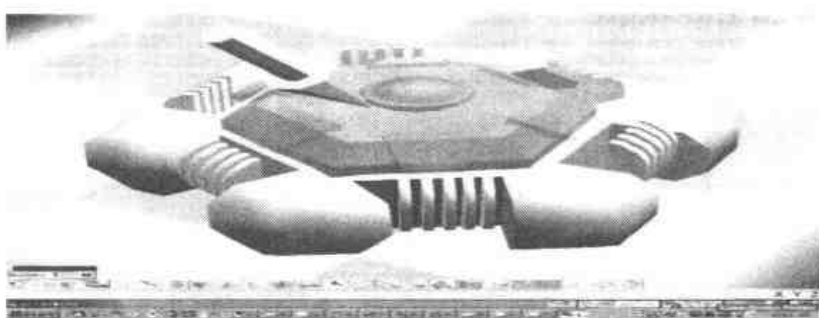
人工智能在游戏中的运用

第十三章

基本物理建模

第十四章

综合运用



11

算法、数据结构、内存管理及多线程

本章将讨论在其他游戏编程参考书中疏漏的细节问题。我们的讨论内容涉及到从游戏的编写到演示的制作以及优化理论。本章将帮您掌握这些必需的编程细节。下一章，我们在讨论人工智能时，读者可以掌握一些 3D 运算，这样便对一般的游戏编程概念有了基本的了解。

本章主要有以下内容：

- ✦ 数据结构
- ✦ 算法分析
- ✦ 优化理论
- ✦ 运算技巧
- ✦ 混合语言编程
- ✦ 游戏的保存
- ✦ 多人游戏的实现
- ✦ 多线程编程技巧

数据结构

在游戏编程中，“游戏所采用的数据结构是什么？”或许是人们最经常提到的问题。答案就是：最快速有效的数据结构。然而，在大多数情况下，你并不需要采用计算机科学所能提供的最先进复杂数据结构。相反，你应该尽可能将其简化。在速度比内存更重要的今天，我们宁可先牺牲内存！

记住这一点，我们先来看一看游戏中最常用的数据结构，并了解一下何时采用以及如何使用这些数据结构。

静态结构和数组

所有数据结构的最基本要素当然是数据项的单独事件，如一个结构体或类。如下例所示：

```
typedef struct PLAYER_TYP // tag for foreword references
{
    int state; // state of player
    int x, y; // position of player
    //...
} PLAYER, *PLAYER_PTR;
PLAYER player_1, player_2; // create a couple player
```

C++

提示 在 C++ 中，你不必使用 typedef 来定义一个结构类型；当你使用到关键字 struct 时，编译器会自动为之创建一个类型。此外，C++ 结构可以包含方法甚至公有和私有部分。

在这个例子中，一个数据结构带有两个静态定义的记录参数来完成工作。另一方面，如果游戏玩家多于三个，那么最好的做法是采用如下所示的数组：

```
PLAYER players[MAX_PLAYERS]; // the players of the game
```

这样，你便可以用一个简单的循环来处理所有的游戏玩家了。OK，太棒了！但是如果游戏运行以前你不知道会有多少玩家或记录参数，那又该如何呢？

当出现这种情况时，我们应计算出数组所应包含元素的最大值。如果它是一个相当小的数值，如 256 或更小，并且每个元素也相当的小（少于 256 字节），我们通常采用静态分配内存，并使用一个计数器来计算任何时候需要激活的元素数目。

你也许觉得这对于内存而言是一种浪费。但是它比遍历一个链表或动态结构提要容易和快速得多。关键在于，如果你在游戏运行前知道数组元素的数目，并且数目不是太大，那么就在游戏启动时通过调用函数 malloc() 或 new() 来静态地预先分配内存。

警告



不要沉迷于静态数组！例如，假如你的结构体大小为 4KB 而且其数组数可能是从 1 到 256。为防止某些时候数组元素的个数达到 256 而产生溢出错误，采用静态分配内存方法时必须为之分配 1MB 的内存。这时，你显然需要更好的方法——采用链表或动态分配的数组来分配内存，以避免浪费。

链表

对于那些在程序启动或编译时可以预估的、简单的数据结构，数组是最合适的处理方法。但对于那些在运行时可能增大或缩小的数据结构而言，应当使用链表这类形式的数据结构处理方法。图 11.1 表示了一个标准的、抽象的链表结构。一个链表由许多节点构成。每个节点都包含信息和指向表中下一个节点的连接的列表。

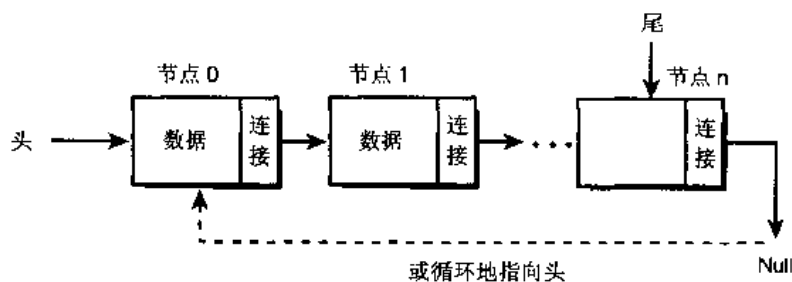
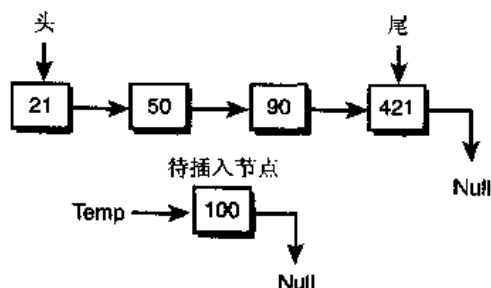


图 11.1 一个链表

链表是很酷的，因为你可以将一个节点插入链表的任意位置。图 11.2 示意了节点插入链表的情形。由于在运行时可以插入或删除节点，使得链表非常适于作为游戏程序的数据结构。

A. 插入前



B. 插入后（依次）

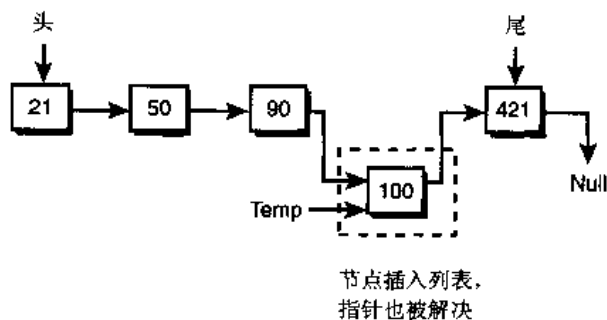


图 11.2 节点插入链表

链表惟一的缺点是你必须一个接一个地遍历节点来寻找你的目标节点（除非创建第二个数据结构来帮助查询）。例如，假定你要定位一个数组中的第 15 个元素，你只需这样便可以访问它：`players[15]`。但对于链表，你需要一个遍历算法以访问链表的节点来定位目标节点。这意味着在最坏的情况下，查询的重复次数与链表的长度相等。这就是 $O(n)$ ——即 n 的最大值。这说明有 n 个元素便要进行 n 次运算。当然，我们可以采用优化算法和第二个包含分类索引链表的数据结构，来达到像访问简单的数组那样快的速度。

创建链表

现在来看一看如何创建一个简单的链表、增加一个节点、删除一个节点以及实现关键字的查询。下面是一个基本节点的定义：

```
typedef struct NODE_TYP
{
    int id;           // id number of this object
    int age;          // age of person
    char name[32];    // name of person
    NODE_TYP *next;  // this is the link to the next node
    // more fields go there
} NODE, *NODE_PTR
```

在建立链表时，需要一个头指针和一个尾指针分别指向链表的头节点和尾节点。开始时链表是空的，因而头尾指针均指向 `NULL`：

```
NODE_PTR head = NULL,
tail = NULL;
```

注 意



有些程序员喜欢以一个空节点作为一个链表的开始节点。这通常是一种个人喜好。但这会改变链表节点创建、插入和删除的算法。你也不妨试一试。

遍历一个链表

出人意料的是，遍历链表是所有链表操作中最容易实现的。

1. 从头指针处开始。
2. 访问节点。
3. 连接到下一节点。
4. 如果指针非指向空（`NULL`），重复第 2 步。

下面是源代码：

```
void Traverse_List(NODE_PTR head)
{
```

```

// this function traverses the linked list and prints out
// each node

// test if head is null
if (head==NULL)
{
    printf("\nLinked List is empty!");
    return;
} // end if

// traverse while nodes
while (head!=NULL)
{
    // visit the node, print it out, or whatever...
    printf("\nNode Data: id=%d", head->id);
    printf("\nage=%d, head->age);
    printf("\nname=%s\n", head->name);

    // advance to next node (simple!)
    head = head->next;
} // end while

print("\n");

} // end Traverse_List

```

哈！非常酷！下一步，让我们看一看如何在链表中插入一个节点。

插入节点

插入节点的第一步是创建该节点。创建节点有两种方法：你可以将新的数据元素传递给插入函数，由该函数来构造一个新节点；或者先构造一个新节点，然后将它传递给插入函数。这两种方法在本质上是相同的。

此外，还有许多方法可以实现节点插入链表的操作。粗糙的做法是将要插入的节点插在链表的开头或结尾。如果你不关心该节点的顺序，这不失为一个便捷的方法。但如果想保持链表原来的排序，你就应当采用更为智能的插入算法。这样可以保证插入节点后的链表仍然保持升序或降序的顺序。这也可以为以后的查询带来更快的速度。

为简明起见，我举一个最简单的节点插入方法，并将该节点插入链表的末尾。其实按顺序的节点插入算法并不复杂。首先要扫描整个链表，找出新节点所要插入的位置，然后将其插入。惟一的问题就是保证指针的跟踪和不丢失任何信息。

下面的源代码是将一个新节点插入该链表的尾部（比插入链表头部难度稍大）。注意一下特殊的情况：空链表和只有一个元素的链表。

```
// access the global head and tail to make code easier
// in real life, you might want to use ** pointers and
// modify head and tail in the function ???

NODE_PTR insert_Node(int id, int age, char *name)
{
    // this function inserts a node at the end of the list
    NODE_PTR new_node = NULL;

    // step 1: create the new node
    new_node = (NODE_PTR)malloc(sizeof(NODE)); // in C++ use new operator

    // fill in fields
    new_node->id = id;
    new_node->age = age;
    strcpy(new_node->name, name); // memory must be copied!
    new_node->next = NULL; // good practice

    // step 2: what is the current state of the linked list?

    if (head == NULL) // case 1
    {
        // empty list, simplest case
        head = tail = new_node;

        // return new node
        return(new_node);
    } // end if
    else
    if ((head != NULL) && (head == tail)) // case 2
    {
        // there is exactly one element, just a little
        // finesse...
        head->next = new_node;
        tail = new_node;

        // return new node
        return(new_node);
    } // end if
    else // case 3
    {
        // there are 2 or more elements in list
        // simply move to end of the list and add
        // the new node
        tail->next = new_node;
        tail = new_node;

        // return the new node
        return(new_node);
    }
}
```

```

    } // end else

} // end Insert_Node

```

你可能觉得代码比较简单。但实际上它却很容易造成混乱，因为它处理的是指针，所以要小心谨慎！聪明的程序员头脑一转便会很快地意识到 case2 和 case3 可以合而为一，而且代码也并不复杂。下面我们来看一看节点的删除。

删除节点

删除节点比插入节点复杂，因为指针和内存都要重新定位、分配。大多数情况下，只需删除一个特殊的节点。这个节点的位置可能是头部、尾部或中间，因此你必须编写一个通用的算法来处理所有可能的情况。如果你没有将所有的情況考虑进去并进行测试，那后果将是非常糟糕的！

一般而言，这个算法必须能够按所给定的关键字搜索链表，删除节点并释放其占用的内存。此外，该算法还必须能够处理指向被删除节点的指针和该节点所指向的下一个节点的指针。看一看图 11.3 便会一目了然。

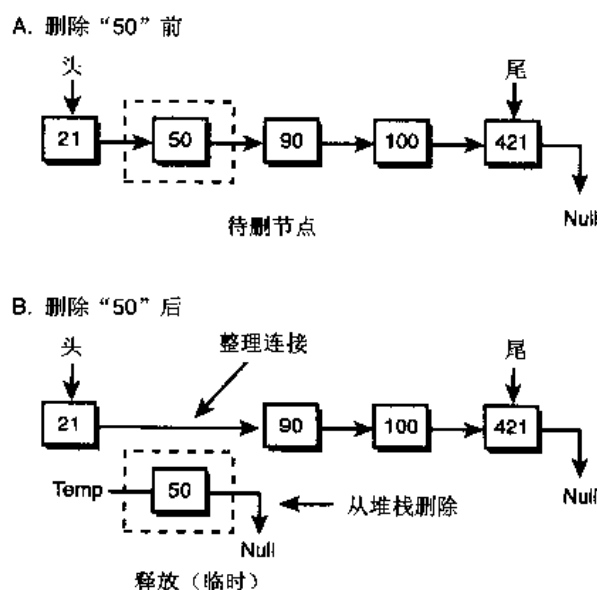


图 11.3 从链表中删除节点

下一段代码可以处理任何情况下基于关键字标识符的节点删除操作：

```

// again this function will modify the globals
// head and tail (possibly)

int Delete_Node(int id) // node to delete
{

```

```

// this function deletes a node from
// the linked list given its id
NODE_PTR curr_ptr = head, // used to search the list
prev_ptr = head; // previous record

// test if there is a linked list to delete from
if (!head)
    return(-1);

// traverse the list and find node to delete
while(curr_ptr->id != id && curr_ptr)
{
    // save this position
    prev_ptr = curr_ptr;
    curr_ptr = curr_ptr->next;
} // end while

// at this point we have found the node
// or the end of the list
if (curr_ptr == NULL)
    return(-1); // couldn't find record

// record was found, so delete it, but be careful,
// need to test cases
// case 1: one element
if (head==tail)
{
    // delete node
    free(head);

    // fix up pointers
    head=tail=NULL;

    // return id of deleted node
    return(id);
} // end if
else // case 2: front of list
if (curr_ptr == head)
{
    // move head to next node
    head=head->next;

    // delete the node
    free(curr_ptr);

    // return id of deleted node
    return(id);
} // end if
else // case 3: end of list

```

```

if (curr_ptr == tail)
{
    // fix up previous pointer to point to null
    prev_ptr->next = NULL;

    // delete the last node
    free(curr_ptr);

    // point tail to previous node
    tail = prev_ptr;

    // return id of deleted node
    return(id);

} // end if
else // case 4: node is in middle of list
{
    // connect the previous node to the next node
    prev_ptr->next = curr_ptr->next;

    // now delete the current node
    free(curr_ptr);

    // return id of deleted node
    return(id);

} // end else

} // end Delete_Node

```

注意代码中包括了许多特殊的情况。每一种情况的处理都很简单，但我还是希望读者能够考虑周全，不放过每一细节！

最后，你或许已经注意到删除链表内部节点的操作极富戏剧性。这是因为这个节点一旦被删除，就无法恢复。我们不得不跟踪前一个 NODE_PTR 以跟踪末尾的节点。这个问题及其他类似问题可以使用如图 11.4 所示的双向链表而得到解决。

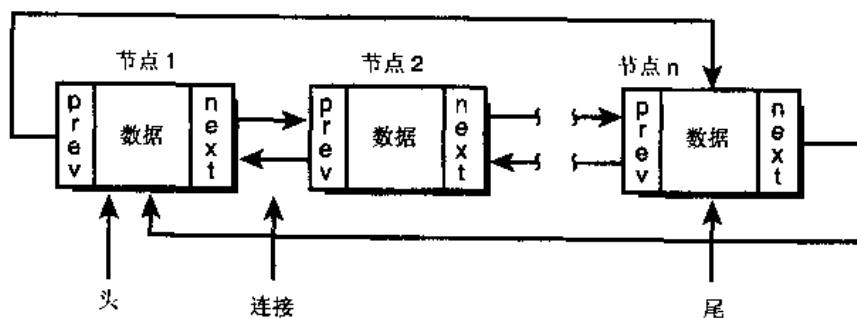


图 11.4 双向链表

双向链表的优点在于你可以在任何位置从两个方向遍历链表节点，可以非常容易地实现节点的插入和删除。数据结构的惟一改变就是另一个连接字段。如下所示：

```
typedef struct NODE_TYP
{
    int id;           // id number of this object
    int age;          // age of person
    char name[32];    // name of person
    // more fields go there
    NODE_TYP *next;   // link to the next node
    NODE_TYP *prev; // link to previous node
} NODE, *NODE_PTR;
```

应用双向链表，你可以从任一个节点向前或向后搜索，所以节点插入和删除所用的跟踪节点大大简化。控制台程序 DEMO11_1.CPP/EXE 便是一个简单的链表操作程序。它可以实现节点的插入、删除和遍历。

注 意

DEMO11_1.CPP 是一个控制台程序而非标准的 Windows .EXE 程序。所以在编译前应将编译器设定为控制台应用程序。当然无需 DirectX，不必加载任何 DirectX .LIB 文件。

算法分析

算法设计与分析通常是高级的计算机科学知识。但我们至少要掌握一些常用的技巧和思想，以利于我们编写比较复杂的算法。

首先，一个好的算法比所有的汇编语言或优化器更好。前面说过，调整一下数据的顺序便能够减少按元素的幅度搜索数据所花费的时间。因此所应掌握的原则是：选择一个可靠的、可以解决和处理数据的算法，与此同时，挑选一种易于该算法访问和操作的的数据结构。

例如，假如你总是使用线性的数组，你就不要指望进行快速的搜索（除非你使用第二个数据结构）。但如果使用排序的数组，搜索时间就会以对数级缩短。

编写一个好算法的第一步是掌握一些算法分析知识。算法分析技术又叫渐近分析，通常是基于微积分的。我不想过多地深入，所以只介绍一些概念。

分析一个算法的基本思想是看一看 n 个元素时主循环的执行次数。这里 n 可代表任何数据结构的元素数目。这是算法分析最重要的思想。当然，有了好的算法后，每次的执行时间、初始化的时间也同样重要，但我们从循环执行的次数开始。让我们看一看下面两个例子：


```

for (int index=0; index<n; index++)
{
    // do work, 50 cycles
} // end for index

```

在这个例子中，程序执行 50 次整数循环，因此执行时间就是 n 的阶数即 $O(n)$ 。它是 $O(n)$ 的上限，也是对执行时间的一个粗糙的上限估计。假如要更精确一点，你知道其内部的计算需要 50 周的时间，所以整个执行时间就是：

$n \times 50 \text{ cycles}$

对吗？错了！如果要计算循环时间，你应当将循环自身花费的时间包括进去。这些时间包括参数的初始化，比较、增量和循环的跳转。将这些时间加进去，如下式所示：

$\text{Cycles}_{\text{initialization}} + (50 + \text{Cycles}_{\text{inc}} + \text{Cycles}_{\text{comp}} + \text{Cycles}_{\text{jump}}) \times n$

上式是一个较好的估计。这里， $\text{Cycles}_{\text{inc}}$ 、 $\text{Cycles}_{\text{comp}}$ 和 $\text{Cycles}_{\text{jump}}$ 分别代表增量、比较和跳转所需的周数。在奔腾级的处理器上，其值约为 1~2 周。因此，在这种情况下，循环本身所花费的时间和程序内部工作循环所需用的时间一样多。这一点是非常重要的。

比如，许多游戏程序员在处理有关像素绘图的问题时，将其编写成函数而非一个宏或内联代码。因为像素绘图函数非常简单，而调用这个函数比直接画图所需时间还要多！所以在设计循环时必须确保循环内部有足够的工作，而且循环运行所需时间要远大于循环自身所花费的时间。

现在让我们来看一看另一个例子——一个比 n 个元素花费更多运行时间的程序：

```

// outer loop
for (i = 0; i < n; i++)
{
    // inner loop
    for (j=1; j<2*n; j++)
    {
        // do work
    } // end for j
} // end for i

```

在这个例子中，我们假定循环中工作部分执行所需时间远大于实现循环机制所花费的时间，这样我们就不考虑循环自身花费时间而只考虑循环执行的次数。这个程序的外部循环次数为 n 次，内部循环的次数为 $2n-1$ 次，所以内部代码总的执行次数为：

$n(2n-1) = 2n^2 - n$

上式由两项构成。 $2n^2$ 是主项，其值要远大于后一项，并且随 n 取值的增加，两项的差值也增大。如图 11.5 所示。

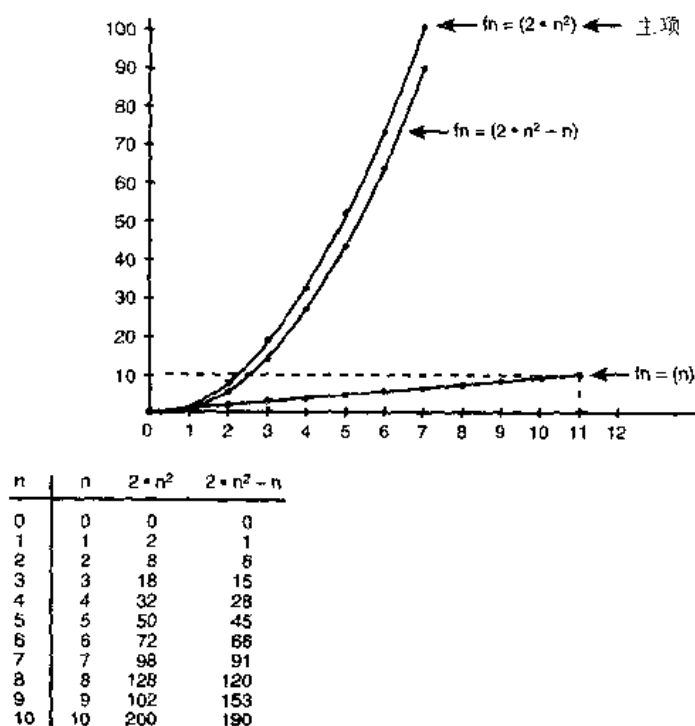


图 11.5 $2n^2 - n$ 的增长速率

当 n 较小比如 $n=2$ 时，上式的值为：

$$2 \times (2)^2 - 2 = 6$$

在该情况下， n 这项减去总花费时间的 25%。但当 n 的值增大时，比如 $n=1000$ ，

$$2 \times (1000)^2 - 1000 = 1999000$$

这时， n 这项只减去总花费时间的 0.5%，可以忽略不计。现在读者该明白了为什么 $2n^2$ 项是主项或更简单地说 n^2 是主项。所以，这时的算法规则是 $O(n^2)$ ，这是非常糟糕的，以 n^2 运算的算法是不能令人满意的，因此当你提出这样一个算法时，你最好从头再来！

上述都是为渐近分析作准备的。最低要求是：你必须能够粗略地估计你的游戏程序中循环的执行时间。这将有助于你挑选算法和所需编码空间。

递归

我们下一个所要探讨的主题是递归。递归是一种应用归纳的方法求解问题的技术。递归的基本含义是把许多问题连续分解成同一形式的简单问题，直到能够真正的求解为止。然后将这些小问题进行归纳、组合进而使整个问题得到解决。听起来是不是很美妙？

在计算机编程中，我们通常使用递归算法来实现搜索、排序和一些数学计算。递归的前提非常简单：编写一个函数，该函数具有调用自己来求解问题的能力。是不是听起来不可思议？其关键在于该函数调用自己时，就在堆栈里创建一组新的局部变量，所以相当于一个新的函数被调用。你惟一需要注意的是函数不能溢出堆栈，程序结束时要有结束处理代码以保证堆栈通过 `return()` 释放空间。让我们看一个标准的实例：阶乘的计算。一个数的阶乘写作 $n!$ ，其含义如下：

$$n! = n(n-1)(n-2)(n-3)\dots(2)(1)$$

并有： $0! = 1! = 1$

这样， $5!$ 就是 $5 \times 4 \times 3 \times 2 \times 1$ 。

以下是用一般的方法编写的计算代码：

```
// assuming intergers
int FACTORIAL(int n)
{
    int sum = 1; // hold result

    // accumulate product
    while (n > 1)
    {
        sum *= n;

        // decrement n
        n--;
    } // end while

    // return the result
    return(sum);

} // end Factorial
```

看上去非常简单。如果输入 0 或 1 则即算结果为 1。若输入 3，则其计算顺序如下：

$$\text{sum} = \text{sum} \times 3 = 1 \times 3 = 3$$

$$\text{sum} = \text{sum} \times 2 = 3 \times 2 = 6$$

$$\text{sum} = \text{sum} \times 1 = 6 \times 1 = 6$$

显然，计算结果正确无误。因为 $3! = 3 \times 2 \times 1$ 。

以下是采用递归方法编写的程序：

```
int Factorial_Rec(int n)
// test for terminal cases
if (n==0 || n==1) return(1);
else
    return(n*Factorial_Rec(n-1));

} // end Factorial_Rec
```

这个程序并不怎么复杂。我们看看当 n 分别为 0 和 1 时，程序是如何运行的。在这两种情况下，第一个 if 语句为 TRUE，就返回值 1 并退出程序。但当 $n>1$ 时，奇妙的事情就发生了。这时，执行 else 语句并返回该函数调用自身 ($n-1$) 次后的值。这就是递归过程。

当前函数变量的取值仍在堆栈里保存，下次调用该函数就相当于调用一个以一组新变量为参数的函数。代码中第一个 return 语句在进行下一个调用前不能执行完毕，就这样一直循环下去直到执行结束语句为止。

让我们来看一看 $n=3$ 时，每次迭代时变量 n 的实际数值。

1. 第一次调用函数 Factorial_Rec(3)，函数开始执行 return 语句：

```
return(3*Factorial_Rec(2));
```

2. 第二次调用函数 Factorial_Rec(2)，函数又开始执行 return 语句：

```
return(2*Factorial_Rec(1));
```

3. 第三次调用函数 Factorial_Rec(1)，这次函数执行结束语句并返回值 1：

```
return(1);
```

现在奇妙的是，1 被返回给第二次调用的函数 Factorial_Rec()，如下所示：

```
return(2*Factorial_Rec(1));
```

这也就计算出了 $\text{return}(2 \times 1)$ 的数值。随之这一数值便被返回给第一次调用的函数，如下所示：

```
return(3*Factorial_Rec(2));
```

这也就计算出了 $\text{return}(3 \times 2)$ 的结果，随之函数最终得到返回值 6 即 $3!$ 。这便是递归过程。这时你会问哪种方法更好一些呢——是递归法还是非递归法？很明显，直接方法执行速度要快一些，因为没有涉及到任何函数调用或堆栈操作。但递归方法更优雅、能更好地反映问题的本质。这就是我们使用递归的原因。有些算法本质上是递归的，为之编写非递归算法会非常冗长，而且最终也必须使用堆栈进行递归模拟。如果适于简化问题，就使用递归法编程；否则就使用直接方法。

例如，看一下程序 DEMO11_2.CPPIEXE。该程序实现了递归算法。注意一下阶乘的溢出速度。看看你的机器能计算到多大阶乘。绝大多数的机器可以计算到 $69!$ 。

数 学

我们来递归一下 Fibonacci 算法。第 n 个 Fibonacci 数列元素 $f_n = f_{n-1} + f_{n-2}$ ，另外， $f_0=0$ ， $f_1=1$ ，那么， $f_2=f_1+f_0=1$ ， $f_3=f_2+f_1=2$ 。所以整个 Fibonacci 序列为：0，1，1，2，3，5，8，13...。数一数向日葵中一圈圈的种子数目，恰好是 Fibonacci 序列。

树结构

树结构是另一类先进的使用递归处理的数据结构。这也是我在上面论述中偏重递归的原因所在。图 11.6 是一些各不相同的树状数据结构图。

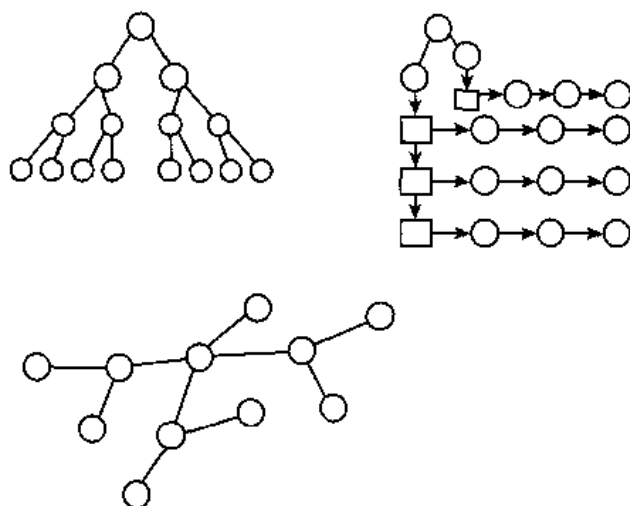


图 11.6 一些树结构的布局

树结构被人们发明出来用于储存和搜索海量的数据。最常见的树结构是二叉树，如 AKA 二叉树 BST 或二叉搜索树——一种从单一根节点发出许多子节点的树状结构。每一节点向下遗传有一个或两个子节点（兄弟）。这就是二叉树的来历。而且我可以讨论一个二叉树的次序和层次。图 11.7 所示的是不同层次二叉树的次序和层次。

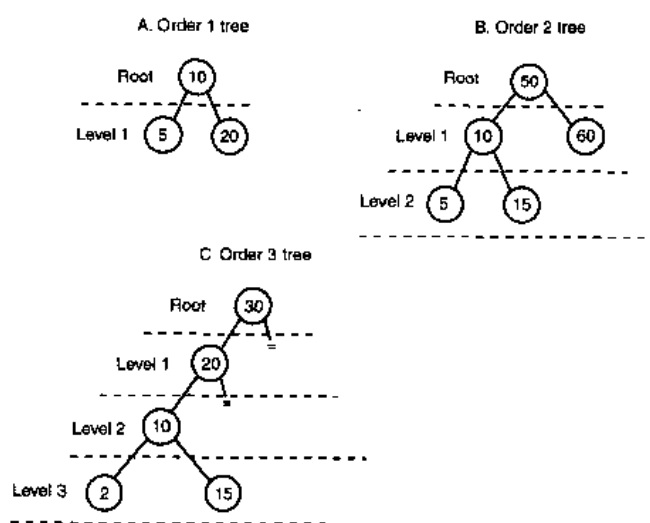


图 11.7 一个二叉树结构及其次序

信息查询的速度是树结构最有意义的问题。绝大多数二叉树使用单一的关键字来查询数据。例如，假定你想创建一个包含游戏对象记录的二叉树，而每一个游戏对象具有多个属性，你可能会使用游戏对象的创建时间作为关键字，或将数据库中的每一节点定义为代表一个人。

下面是可以用来控制个人节点的数据结构：

```
typedef struct TNODE_TYP
{
    int age;           // age of person
    char name[32];     // name of person
    TNODE_TYP *right;  // link to right node
    TNODE_TYP *left;   // link to left node
} TNODE, *TNODE_PTR
```

注 意

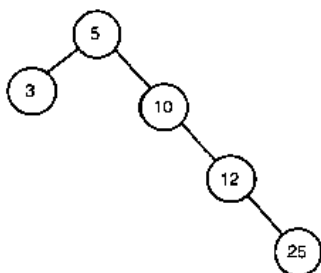


在本例中，数据可以定义为任何值。

注意树节点与链表节点的相似之处。它们之间惟一的不同之处是树节点的建立及其数据结构使用方法。看一看上例，假如有五个对象，其年龄分别为 5、25、3、12 和 10。图 11.8 表示包含该数据的两个不同的二叉树。包含这些属性的二叉树的创建方法取决于数据在插入算法中的次序。

注意在建立如图 11.8 所示的二叉树时，使用了如下约定：右边的子节点总是大于或等于该节点的父节点，左边的子节点总是小于其父节点。你还可以使用其他的约定并恪守它。

A. 一种可能的树



B. 另一种可能的树

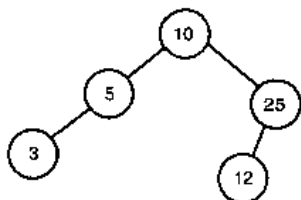


图 11.8 数组年龄 [5, 25, 3, 12, 10] 的 B 树结构编码

二叉树可以存储海量数据，而且应用“折半检索法”可以快速地检索数据。这是二叉树的优越性所在。比如，如果一个二叉树带有 1 兆节点，至多进行 20 次比较便可以检索到数据！这是不是太疯狂？其原因在于检索中的每次迭代便去除搜索空间中一半节点。从根本上说，假如有 n 个节点，平均搜索次数为 $\log_2 n$ ，运行时间为 $O(\log_2 n)$ 。

注 意

上述所说的搜索时间只适于均衡二叉树——二叉树的每一层次均含有相等的左右子节点。如果二叉树不是均衡的，那么它就退化为一个链表，而搜索时间也退化为一个线性函数。

二叉树第二个非常酷的属性是你可以定位子树并单独处理它。该子树仍然具有二叉树的所有属性。因此，如果你知道检索的位置，便可以搜索子树检索你所需要的东西。这样一来，你便可以创建树结构的数结构或建立子树的索引表，而不必处理整个树。这在 3D 建模中是非常重要的。你可以为整个游戏空间建立一个树结构，而以成百上千个子树来表示空间中的各个场所。你还可以创建另一个树结构来指代一个空间顺序的指针链表。该链表指针指向子树的节点，如图 11.9 所示。有关这方面更多的内容会在本书的以后章节中论及。

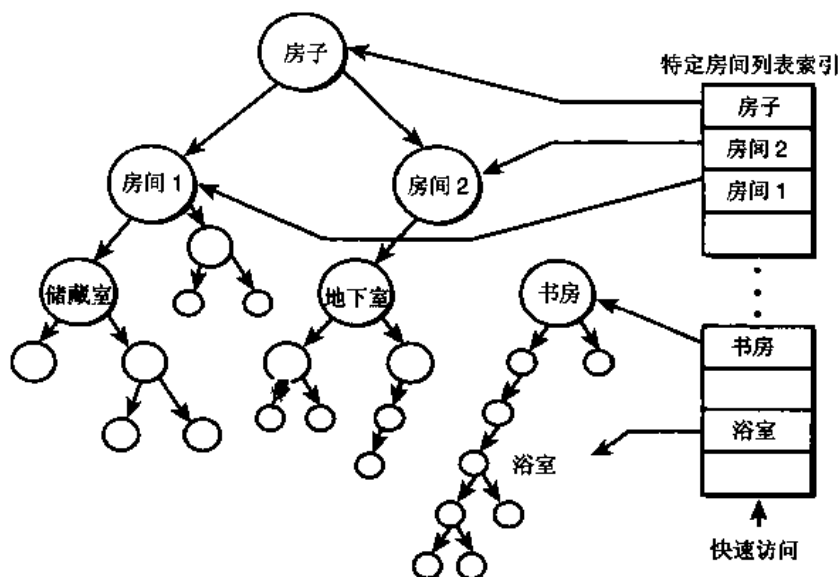


图 11.9 在 B 树结构上使用一个二级索引表

最后，让我们谈一谈何时使用树结构。我认为当所处理的问题或数据类似于以树结构开始时适于使用树结构。如果你自己将问题抽象出来并发现其左右分支，很明确，你应采用树结构。

建立 BST

由于应用于二叉树的算法在本质上是递归的，所以这一节的主题比较复杂。让我们快

速地浏览一下二叉树的一些算法、节点的编写，并完成一个演示程序。

与链表相似，创建 BST 有两种方法：树结构的根节点可以为空或实节点。我更喜欢以实节点作为根节点。因此，一个空树中没有任何东西，而其根节点指向 NULL。

```
TNODE_PTR root = NULL;
```

OK！要将数据插入 BST，你必须确定插入数据所使用的关键字。在本例中，你可以使用人的年龄或名字。因为这些例子都使用年龄作为关键字，所以这里也使用年龄作为关键字。然而，以名字作为关键字更简单，你只需使用字典式的比较函数如 `strcmp()` 来确定名字的顺序。下面是将节点插入 BST 的通用代码：

```
TNODE_PTR root = NULL; // here's the initial tree

TNODE_PTR BST_Insert_Node(TNODE_PTR root, int id, int age, char *name)
{
    // test for empty tree
    if (root==NULL)
    {
        // insert node at root
        root = new(TNODE);
        root->id = id;
        root->age = age;
        strcpy(root->name,name);

        // set links to null
        root->right = NULL;
        root->left = NULL;

        printf("\nCreating tree");

    } // end if

    // else there is a node here, lets go left or right
    else
    if (age >= root->age)
    {
        printf("\nTraversing right...");
        // insert on right branch

        // test if branch leads to another sub-tree or is terminal
        // if leads to another subtree then try to insert there, else
        // create a node and link
        if (root->right)
            BST_Insert_Node(root->right, id, age, name);
        else
        {
            // insert node on right link
            TNODE_PTR node = new(TNODE);
```



```

    node->id    = id;
    node->age    = age;
    strcpy(node->name, name);

    // set links to null
    node->left    = NULL;
    node->right    = NULL;

    // now set right link of current "root" to this new node
    root->right = node;

    printf("\nInserting right.");

    } // end else

    } // end if
else // age < root->age
    {
        printf("\nTraversing left...");
        // must insert on left branch

        // test if branch leads to another sub-tree or is terminal
        // if leads to another subtree then try to insert there, else
        // create a node and link
        if (root->left)
            BST_Insert_Node(root->left, id, age, name);
        else
        {
            // insert node on left link
            TNODE_PTR node = new(TNODE);
            node->id    = id;
            node->age    = age;
            strcpy(node->name, name);

            // set links to null
            node->left    = NULL;
            node->right    = NULL;

            // now set right link of current "root" to this new node
            root->left = node;

            printf("\nInserting left.");
        } // end else

    } // end else

// return the root
return(root);

} // end BST_Insert_Node

```

首先你要测定空树，然后创建其根节点。如有必要，应使用第一项内容创建根节点。这样，被插入 BST 的第一项内容或记录应表示搜索空间靠近中间的项，以便树能很好地平衡。总之，如果树有超过一个的节点，你必须遍历该树，至于将节点插入左子树或右子树则取决于你要插入的记录。当你找到树结构的端节点或结束分支时，你便可以将新节点插入该处。

```
root= BST_Insert_Node(root, 4, 30, "jim");
```

图 11.10 示意了如何将“Jim”插入一个树中。

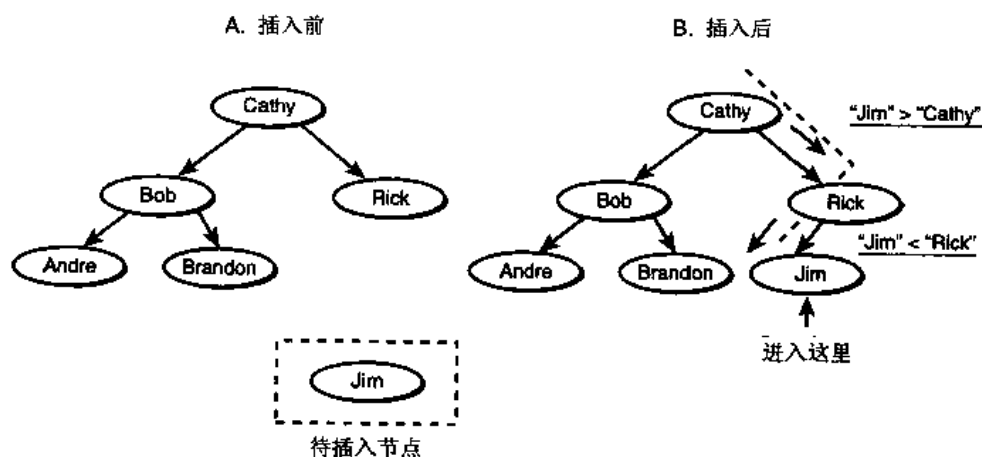


图 11.10 在 BST 中插入

将新节点插入 BST 这一过程的执行性能和查询该节点相似，因此，一次插入操作所需的平均时间约为 $O(\log_2 n)$ ，而最坏的情况是 $O(n)$ （当关键字以降序线性排列时）。

检索 BST

一旦建立了 BST，数据的搜索是顺理成章的事。当然这会用到许多递归处理方法，应加以关注。BST 搜索的实现有三种方法：

前序——访问节点、按前序搜索左子树，然后按前序搜索右子树。

中序——按中序搜索左子树，访问节点，然后按中序搜索右子树。

后序——按后序搜索左子树，按后序搜索右子树，然后访问节点。

图 11.11 显示了一个简单的二叉树及三种搜索顺序。

注意



左和右是任意的，关键在于访问和搜索的顺序。

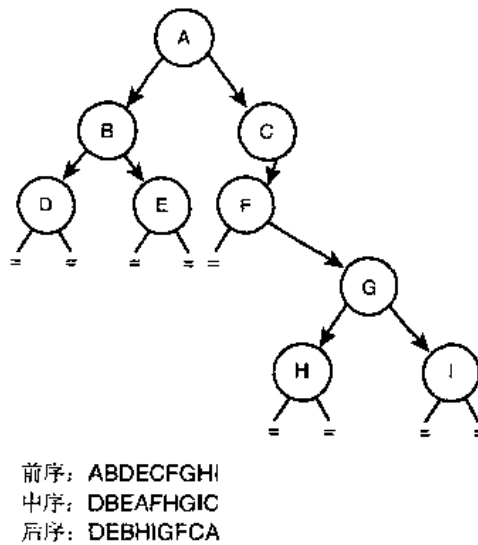


图 11.11 前序、中序、后序搜索方式的节点访问次序

知道了这三种顺序，你可以为之编写简单的递归算法来实现它们。当然，遍历二叉搜索树的目的是找到目标并将其返回。下面的函数便实现了二叉树遍历的功能。你可以为之增加停止代码以便搜索到目标时结束程序。不过，采取何种搜索方法取决于个人的偏好。

```

Void BST_Inorder_Search(TNODE_PTR root)
{
    // this searches a BST using the inorder search

    // test for NULL
    if (!root)
        return;

    // traverse left tree
    BST_Inorder_Search(root->left);

    // visit the node
    print("name:%s, age: %d", root->name, root->age);

    // traverse the right tree
    BST_Inorder_Search(root->right);
} // end BST_Inorder_Search

```

以下是前序搜索代码：

```

Void BST_Preorder_Search(TNODE_PTR root)
{
    // this searches a BST using the preorder search

```

```
// test for NULL
if (!root)
return;

// visit the node
print("name:%s, age; %d", root->name, root->age);

// traverse left tree
BST_Inorder_Search(root->left);

// traverse the right tree
BST_Inorder_Search(root->right);
} // end BST_Preorder_Search
```

以下是后序搜索代码:

```
Void BST_Postorder_Search(TNODE_PTR root)
{
// this searches a BST using the postorder search

// test for NULL
if (!root)
return;

// traverse left tree
BST_Inorder_Search(root->left);

// traverse the right tree
BST_Inorder_Search(root->right);

// visit the node
print("name:%s, age; %d", root->name, root->age);
} // end BST_Inorder_Search
```

就这么简单——像变戏法似的? 如果你建立了一个二叉树, 试着按下述代码进行中序遍历。

```
BST_Indoorder_Search(my_tree);
```

技巧



我很难形容类树结构在 3D 图形中有多重要, 希望你已经理解了上述内容。否则, 当你建立一个空的二叉树来解决所提出的问题时, 你将会陷入指针递归的苦海中。

你注意到我省略了如何删除一个节点。我是有意这样做的。因为删除一个节点非常复杂, 有可能破坏子树的父节点和其与子节点间的连接。唉! 节点删除就留作读者的一个练习好了。这里我向大家建议一本好的参考书——由 Addison Wesley 出版社出版、Sedgewick

撰写的《Algorithms in C++》，这本书深入讨论了树结构及其相关算法。

最后，读者可以检验一下二叉搜索树的一个实例程序——DEMO11_3.CPP1EXE。该程序允许你创建一个二叉搜索树并使用三种算法遍历它。这也是一个控制台程序，因此需要编译它。

优化理论

没有任何程序比游戏程序更需要性能优化。视频游戏总是不断地突破硬件和软件的极限而推动其向前发展，永无止境。游戏编程人员总是希望加入更多的游戏事物、效果和音效，更好的人工智能等等，因此，优化至关重要。

在这一节，我们将讨论一些优化技术以帮助你进行游戏编程。如果你对此有浓厚的兴趣，有很多关于该方面的书可以参考，如山 Addison Wesley 出版社出版、Rick Booth 编著《Inner Loops》，由 Coriols 集团出版、Mike Abrash 编著的《Inner Loops》，AP 出版社出版、Mike Schmit 编著的《Pentium Processor Optimization》。

运用你的智力

编写优化代码的第一要旨是了解编译器、数据结构以及你的 C/C++ 程序最终是如何被转换为可执行的机器代码的。其基本思想是使用简单的程序设计和数据结构。你的代码越复杂、设计越机巧，编译器将其变换为机器代码就越困难，所以其执行速度也就越慢（在大多数情况下）。以下是编程时需要遵循的基本原则：

- 尽可能使用 32 位数据。8 位数据虽然占用空间较少，但英特尔的处理器是以 32 位为基准的，并针对 32 位数据进行了优化。
- 对于频繁调用的小函数而言，应使用内联函数。
- 尽可能使用全局变量，避免产生蹩脚的代码。
- 加法和减法运算应尽量避免浮点数，因为整数单元通常比浮点单元运算快。
- 任何时候都要尽可能使用整数。尽管浮点处理器几乎和整数处理一样快，但整数更精确。所以如果你不需要精确的小数位，就使用整数。
- 将所有的数据结构均调整为 32 位字长，在大多数编译器上你可以使用指令进行手工调整，或在代码中使用 #pragmas。
- 除非是简单类型的参数，否则绝不使用值传递的方式传递参数。
- 在代码中不要使用 register 关键字。尽管 Microsoft 声称它能够加快循环，但这会造成寄存器变量无法编译，并产生错误代码而结束。
- 如果你是位 C++ 程序员，用类和虚函数是可以的。但软件的继承和层次不要太多。

- 奔腾级的处理器使用一个整数数据和代码缓存。了解这一点后，要确保函数代码短小以适应缓存（16KB~32KB 以上）的大小。此外，在储存数据时，将其以可被访问的方式存储。这样可以降低缓存崩溃几率、减少主内存或二级缓存的访问次数。主内存或二级缓存的访问速度要比内部缓存的访问速度慢 10 倍。
- 奔腾机的处理器具有类 RISC 的内核，因此擅长处理简单指令，并允许在一个或多个执行单元期间同时处理二个或多个指令。不要在一行代码中出现晦涩难懂的代码，最好编写较简单的代码，即使你可以将这些代码合并成具有同样功能的一行代码也尽量改成简单的。

数学技巧

由于游戏编程中大量工作的实质是数学问题，因此了解执行数学函数的更好方法是非常值得的。有许多常用的技巧和方法来帮助你达到这一目标：

- 参与整数运算的数必须是整数，参与浮点运算的数必须是浮点数。类型转换必然降低性能。所以除非迫不得已，否则不进行类型转换。
- 通过左移位操作可以实现整数与 2 的任何次幂的乘法运算。同样地，通过右移位操作可以实现整数与 2 的任何次幂的除法运算。对于整数与非 2 的次幂的数的相乘和相除可以转换为和运算或差运算。例如，640 虽不是 2 的整数次幂，但 512 和 128 是 2 的整数次幂；所以当某整数与 640 相乘最好的方法是进行如下变换：

```
product = (N<<7) + (n<<9); // n*128 + n*512 = n*640
```

然而，如果处理器在 1~2 个循环内可以完成乘法运算，这种优化就毫无意义。

- 如果你的算法中应用到矩阵操作，要充分利用矩阵的稀疏性——一些元素为零。
- 在创建常量时，确保其为恰当的类型以防编译器编译出错或将其强迫转换为整数类型。最好的思想是使用新版 C++ 中的 `const` 指令。例如：

```
const float f=12.45;
```

- 避免平方根、三角函数或任何复杂的数学函数。一般而言，这些复杂函数的计算都可以利用特定的假设或近似而找到一个简单的方法来实现。但如果结果更糟糕，你总可以做一个查找表。我在后面将会提及该表。
- 如果你要将一个大的浮点数组清零，要按如下方式使用 `memset()`：

```
memset((voi *) float_array,0,sizeof(float) *num_floats);
```

然而，也只有这时才能这样做，因为浮点数是以 IEEE 格式编码的，而且只有整数和浮点数的数值为零时才相等。

- 在执行数学运算时，看一看在译码前可否手工简化表达式。比如， $n(f+1)/n$ 等于

(f+1)，因为除法和乘法的运算结果将 n 消去了。

- 如果你要执行复杂的数学运算，而且你在下面的代码中需要再次执行它，那么就将其储存在高速缓存中，如下所示：

```
// compute term that is used in more than one expression
float n_squared = n*n;

// use term in two different expressions
pitch = 34.5*n_squared+100*rate;
magnitude = n_squared / length;
```

- 最后，很重要的一点是确保将编译器的选项设定为使用浮点处理器进行编译和创建代码。这样编译的程序尽管不算小，但执行速度快。

定点数学

几年以前，大多数 3D 引擎采用定点数学来完成 3D 中大量的变换和数学运算。这是因为处理器对浮点的支持没有对整数的支持快，甚至在奔腾处理器上也是如此。但是今天，奔腾 II、III 和 Katmai 处理器拥有更好的浮点能力，所以定点运算不再那么重要。

然而在许多情况下，由浮点到整数的光栅变换仍然很慢，因此在内部循环的加法和减法运算中定点运算仍是一个较好的选择。在低档的奔腾机器上，这类运算依然比浮点运算要快，因此你可以运用技巧快速地从定点数中提取出整数部分，而不必进行浮点到整数的转换运算。

无论如何，这些都具有一定的不确定性。今天，使用浮点来处理任何运算通常都是最好的选择。但这一点也不妨碍我们对定点运算的了解。我的观点是使用浮点来处理所有的数据表示和变换，对于低水平的光栅变换可以分别试一试定点运算和浮点运算，看一看那种处理方法最快。当然，如果你使用纯硬件加速，就不用这么做了，只要坚持浮点处理即可。

记住以上所述，下面让我们来看看定点的表示。

定点数的表示

所有的定点数学实际上是以整数尺度为基础的。比如，我们想用—个整数来表示 10.5。这做不到，因为没有小数位。你可以将其截断为 10.0 或将其舍入为 11.0，但 10.5 不是一个整数。但如果你将 10.5 放大 10 倍，10.5 就变成了 105.0，这是一个整数。这便是定点的基础。你可以使用某一系数缩放某数并在进行数学计算时将缩放系统考虑进去。

由于计算机是二进制的，大部分游戏程序倾向于使用 32 位整数以 16.16 的格式来表示定点数。如图 11.12 所示。

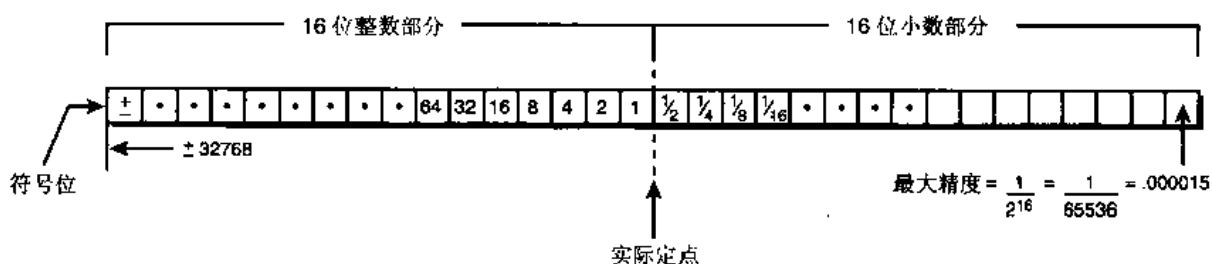


图 11.12 一种 16.16 定点表示方法

你可以将整数部分放在高 16 位，小数部分置于低 16 位。这样你已将整个数值放大为原来的 2^{16} 即 65536 倍。另外，为提取出一个定点数的整数部分，你可以移位、屏蔽掉高 16 位；为得到小数部分，你可以移位、屏蔽掉低 16 位。

以下是一些常用的定点数学形式：

```
#define FP_SHIFT 16      // shifts to produce a fixed-point number
#define FP_SCALE 65536   // scaling factor

typedef int FIXPOINT;
```

定点数的转换

有两类数需要转换为定点数：整数和浮点数。这两类转换是不同的，需分别考虑。对于整数，直接用其二进制的补码表示。所以你可以移位操作来放大这个数，从而将其转换为定点数。而对于浮点数，由于其使用 IEEE 格式，有一个尾数和储存于四字节中的指数，因此移位将破坏这个数。因此，必须使用标准的浮点乘法来进行转换。

数 学

2 的补码是一种表示二进制整数的方法。因此整数和负数均可以用这种方法表示，并且在该集合上数学运算都是正确的。一个二进制数的补码是指将该二进制的每一位取反并与 1 进行加运算所得的数。在数学意义上，假定你想求 6 的补码，先取反得 -6，再与 1 相加即 $-6+1$ 或 $\sim 0110+0001=1001+0001=1010$ ，该数就是 10 的二进制表示，但以 2 的补码表示时是 -6。

以下是整数变换为定点数的宏语句：

```
#define INT_TO_FIXP(n) (FIXPOINT((n<<FP_SHIFT)))
```

例如：

```
FIXPOINT speed = INT_TO_FIXP(100);
```


下面是浮点数变换为定点数的宏语句：

```
#define FLOAT_TO_FIXP(n) (FIXPOINT((float)n *FP_SCALE))
```

例如：

```
FIXPOINT speed = FLOAT_TO_FIXP(100.5);
```

提取一个定点数也很简单。下面是从高 16 位提取整数部分的一条宏语句：

```
#define FIXP_INT_PART(n) (n>>16)
```

至于从低 16 位提取小数部分，则只需将整数部分屏蔽掉即可：

```
#define FIXP_DEC_PART(n) (n & 0x0000ffff)
```

当然，如果你聪明的话，可以忘却变换而使用指针即时地访问高 16 位和低 16 位即可。如下所示：

```
FIXPOINT fp;
short * integral_part = &(fp+2), *decimal_part = &fp;
```

指针 `integral_part` 和 `decimal_part` 总是指向你所需的 16 位数。

精度

此刻一个问题突然出现在你的脑海中：小数部分的精度如何？通常你不必理会这个问题，它仅在运算时用到。一般而言，在光栅转换循环或其他循环中只需整数部分即可。但因为以 2 为基数，所以小数部分也是以 2 为基数的小数。如图 11.12 所示。例如数 1001.0001 是 $9+0\times 1/2+0\times 1/4+0\times 1/8+1\times 1/16=9.0625$ 。

这向我们表明了精度的概念。上式使用了四位二进制数，其精度与 1.5 位十进制数精度或 ± 0.0625 相同。对于 16 位数字，可以精确到 $1/2^{16}=0.000015258$ 或 $1/10000$ 这一精度可以满足绝大部分要求。另一方面，如果你仅用 16 位来存储整数部分，其存储的数值范围为 -32767~32768（无符号数可到 65535）。这在巨大的宇宙或数字空间将成为问题，所以要谨防溢出。

加法和减法运算

定点数的加法和减法运算比较简单。你可以采用标准的+和-运算符：

```
FIXPOINT f1= FLOAT_TO_FIX(10.5),
          f2= FLOAT_TO_FIX(-2.6),
          f3= 0; // zero is 0 no matter what baby

// to add them
```

```
f3=f1+f2;

// to subtract them
f3=f1-f2
```

数 学

由于定点数以二进制的补码表示，所以在进行上述运算时正负数都无问题。

乘法和除法运算

乘法和除法运算比加法和减法运算稍微复杂一些。问题在于定点数是被放大的数。当进行乘法运算时你不仅要乘以定点数还要乘上放大系数。看一看下述源代码：

```
f1=n1*scale
f2=n2*scale
f3=f1*f2=(n1*scale)*(n2*scale)=n1*n1*scale2
```

看到额外的放大系数吗？为矫正这个问题，你需要除以或移出放大系数的平方 $scale^2$ 。这样，两定点数相乘的运算如下：

```
f3=((f1*f2)>>FP_SHIFT);
```

定点数的除法也会遇到同乘法类似的问题，但结果相反。如下所示：

```
f1=n1*scale
f2=n2*scale
```

假设这样，则：

```
f3=f1/f2=(n1*scale) / (n2*scale) = n1/n2 // no scale!
```

注意在运算过程中消去了放大系数因而得到的是非定点数。这在某些情况下是非常有用的。但若要成为定点数，必须进行放大：

```
f3=(f1<<FP_SHIFT) / f2;
```

警 告



定点数的乘法和除法所遇到的问题就是上溢和下溢问题。就乘法而言，最坏的情况是得到 64 位的结果。同样，对于除法分子的高 16 位通常丢失，仅剩下小数部分。解决的方法是使用 24.8 位格式或全 64 位格式进行运算。这可以使用汇编语言实现，因为 Pentium+ 处理器支持 64 位运算。或者，你可以稍微改变一下格式，使用 24.8 位格式。这样可以满足定点数的乘法和除法运算而不丢失任何信息。但你的精度将大幅度下降。

程序 DEMO11_4.CPPIEXE 是一个定点数运算的实例。该程序允许你输入两个小数，然后执行定点数运算并观察其计算结果。注意乘法和除法运算根本不执行，这是因为计算结果采用的是 16.16 格式而非 64 位格式。为修正这一点，你可以采用 24.8 格式重新编译程序。条件编译由两个宏义语句控制：

```
// #define FIXPOINT16_16
// #define FIXPOINT24_8
```

注释掉其中一句，编译器便按剩下的一句进行编译。这是一个控制台应用程序，因此如 Spike 所说，做要做的事……

展开循环

下一个优化技巧是循环展开。在 8/16 位时代，这是最好的优化技术，但今天它可能带来相反的后果。展开循环意味着分解一个重复多次的循环，并对每一行进行译码。举例如下：

```
// loop before unrolling
for (int index=0; index<8; index++)
{
    // do work
    sum+=data[index];
} // end for index
```

这个循环的问题是工作花费的时间小于循环增量、比较和跳转所花费的时间。如此一来，循环本身所需时间是工作代码的二或三倍。你可以进行如下的循环展开：

```
// the unrolled version
sum+=data[0];
sum+=data[1];
sum+=data[2];
sum+=data[3];
sum+=data[4];
sum+=data[5];
sum+=data[6];
sum+=data[7];
```

这样就改进了很多。这里需要说明的有如下两点：

- 如果循环体比循环结构复杂得多，那就没有必要将其展开。例如，如果你要在循环的工作代码内计算平方根，过多的重复操作将无助于代码效率的提高。
- 由于奔腾处理器带有内部缓存，将一个循环展开太多会导致内部缓存的拥塞。这将是灾难性的，并会致使代码异常结束。我的建议是依据情况，循环展开的次数宜为 8~32 次。

查表法

这是我偏爱的优化技巧。查表法是预先计算出程序运行时的一些结果。在程序启动时简单地计算出所有可能的结果，然后再运行游戏。例如，假定你需要计算出从 0~359 间各个角度的正弦和余弦值。如果使用浮点处理器来计算 `sin()` 和 `cos()`，将很费时间。但使用查表方法程序，则只需几个 CPU 周期便可以计算出各角度的正弦和余弦值。举例如下：

```
// storage for look up tables
float SIN_LOOK[360];
float COS_LOOK[360];

//create look-up table
for (int angle=0; angle < 360; angle++)
{
    // convert angle to radians since math library uses
    // rads instead of degrees
    // remember there are 2*pi rads in 360 degrees
    float rad_angle = angle * (3.14159/180);

    // fill in next entries in look-up tables
    SIN_LOOK[angle] = sin (rad_angle);
    COS_LOOK[angle] = cos(rad_angle);
} // end for angle
```

以下是一个使用查表法的例子，在该例中代码执行的结果是画一个半径为 10 的圆：

```
for (int ang = 0; ang<360; ang++)
{
    // compute the next point on circle
    x_pos = 10* COS_LOOK[ang];
    y_pos = 10*SIN_LOOK[ang];
    // plot the pixel
    plot_pixel((int )x_pos+x0,(int)y_pos+y0,color);
} // end for ang
```

当然，查表法需要占用一定的内存，但这样做也是值得的。“如果你能够预先算出结果，就将其放进查询的表中。”这是我的座右铭。你可以思考一下 DOOM、Quake 以及我喜爱的游戏 Half-Life 是怎样运行的？

汇编语言

我想讨论的最后一个优化技术是汇编语言。你或许已拥有了最佳的算法和优秀的数据结构，但你还是希望数据位多一些、代码运行更快一些。手工汇编语言不可能使代码如同运行在 8/16 位处理器上那样快 1000 倍，但它能够使你的代码运行速度提高 2~10 倍。这说

明使用手工汇编语言也是非常值得的。

当然，你必须确保只转换游戏程序中需要转换的代码部分。注意不要将主程序搞乱，因为那样会浪费时间。用一个探测器查看一下所有游戏程序的 CPU 循环在何处被消耗殆尽（可能在图形部分），然后定位该处将其转换为汇编语言。我建议使用 Intel 的 Vtune 作为查看器。

在过去（几年前），大部分编译器不具备内联汇编器。如果有，也是极少数！如今 Microsoft、Borland 和 Watcom 的汇编器与它们的编译器内联在一起。将其用于从数十句到几百句的小程序就如同单独使用汇编器一样得心应手。所以我建议如果需要汇编语句就使用内联的汇编器。下面的代码表明在使用 Microsoft VC++ 时如何激活内联的汇编器：

```
_asm
{
    .. assembly language code here
} // end asm
```

内联汇编器最突出的优点是它允许使用已在 C/C++ 中定义的变量名。下面的代码表明了如何使用内联的汇编语言编写一个 32 位的内存填充函数：

```
void qmemset(void *memory, int value, int num_quads)
{
    // this function uses 32 bit assembly language based
    // and the string instructions to fill a region of memory
    _asm
    {
        CLD          // clear the direction flag
        MOV EDI, memory // move pointer into EDI
        MOV ECX, num_quads // ECX hold loop count
        MOV EAX, value   // EAX hold value
        REP STOSD        // perform fill
    } // end asm

} // end qmemset
```

要使用这个函数，你只需执行下述代码：

```
qmemset(&buffer, 25, 1000);
```

这样从缓冲的起始地址填充值 25 开始，1000 quads 便被逐一填充。

注 意



如果你使用的不是 Microsoft VC++，你应查看一下你所用编译器的帮助，弄明白内联汇编器所需的语法格式。在大多数情况下，只不过是些零零落落的下划线的区别。

制作演示程序

假如你已完成游戏程序的编写，这时需要一个演示模式。制作演示程序主要有两种方法：你可以自己玩这个游戏并记录你的动作，或者你可以使用一个人工智能玩家。记录自己的游戏玩法是最常见的选择。因为编写一个像真人一样过关斩将的人工智能玩家是非常困难的，而且为了给潜在的买家留下良好的印象，就必然要求人工智能玩家以非常酷的方式玩游戏，要做到这一点也是很困难的。让我们扼要地看一下这两种方法是怎样实现的。

预先记录的演示程序

为记录演示程序，基本而言，你要记录每一循环的各种输入设备的状态，将数据写入文件，然后将该记录文件作为游戏引擎的输入制作演示程序。看一看图 11.13A 及 B 便一目了然。这一方法的思想是游戏本身并不知道数据的输入是来自键盘（输入设备）或文件，因此这种演示程序只是简单地播放游戏。

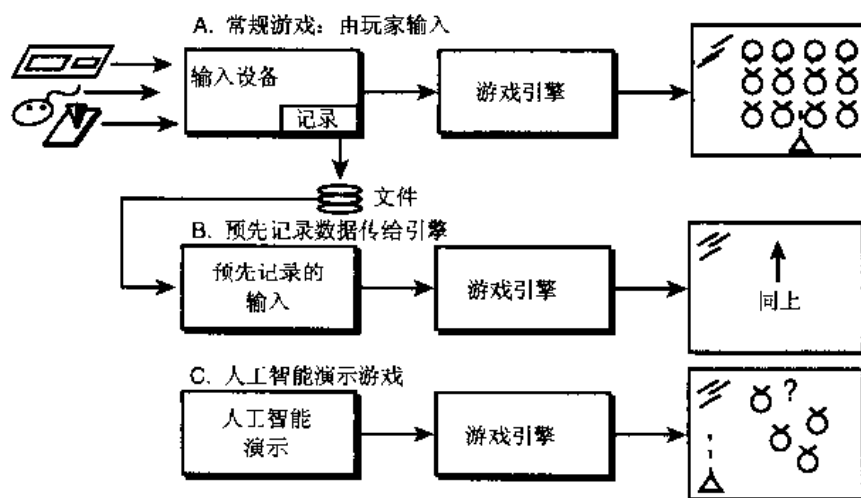


图 11.13 演示程序回放

为使其工作，你必须有一个确定性的游戏策略：如果你再次玩这个游戏并且玩法相同，那么游戏人物也将做同样的事情。这意味着如同记录输入设备一样，你必须记录初始的随机数，以便将一个记录的游戏的开始状态像输入一样地记录下来。这样做确保了演示游戏按照你所记录的状态播放。

记录一个游戏的最好途径不是以一定的时间间隔对输入进行采样，而是对每一帧的输入进行采样。这样一来，这个演示游戏不论是运行于慢的机器上还是快的机器上，播放数据均能与游戏保持同步。我通常的做法是将所有的输入设备并入到一个记录中，一帧一个记录，然后将这些记录做成一个文件。将播放演示程序的状态信息放在文件的开头，以便

于这些数据的回复加载。因此，这个播放文件的形式如下所示：

```
initial state information

Frame 1: Input Values
Frame 2: Input Values
Frame 3: Input Values
.
.
Frame N: Input Values
```

一旦你完成了这个文件，只要简单地将游戏重置、倒回播放即可。随后的读入文件如同由输入设备输入数据一样。游戏自身并不知道这些差别！

警告



你可能出现的一个最糟糕的错误是在写出记录时对输入采样的时机不当。你必须确保你所采样和记录的输入是游戏相应的帧所对应的输入。一个新手常犯的错误是在为游戏演示所进行的采样常超前或落后于游戏的正常输入时刻。因此，所采样的数据不是所需的输入数据！其可能造成的结果是游戏玩家在游戏事件循环的某一部分已有了发射键，而在另一部分却没有。所以为了游戏正常的运行，采样时必须做到采样与读入的输入同步。

人工智能控制的演示程序

记录游戏的第二个方法是借助于编写的人工智能“bot”来执行游戏，就像人们联网玩 Quake 一样。bot 在游戏处于演示模式时开始玩游戏就如同一个人工智能角色参与游戏一样。这种方法惟一的问题（除技术复杂外）是 bot 可能没有卖弄所有的“酷”的房间、武器等等，因为它并不知道它在制作游戏的演示。另一方面，采用 bot 参与游戏的最大好处是每一个演示都不相同，并且不会将最具吸引力的玩法泄露出来。

在游戏中使用 bot 和使用其他的人工智能人物如出一辙。基本上你只需将其与你的游戏输入接口连接起来并重载标准输入流即可，如图 11.13C 所示。然后为 bot 编写人工智能算法，设定一些主要的目标，如找出迷宫的路径，射杀所见的每一个东西或其他任务等。此后非常简单，你只需放任 bot 去演示游戏，直到游戏爱好者对其产生兴趣。

保存游戏的策略

在游戏编程中，游戏保存部分的编写是最令人头疼的事情之一。这是游戏程序员最后要做并为之殚精竭虑的事情之一。其核心是为你所编写的游戏的玩家提供保存游戏进度的功能。

在任何时候都能保存游戏意味着要记录游戏中每一个变量和每一个对象。因此在一个

文件中，你必须记录所有的全局变量和每个对象的状态。最佳的实现途径是采用面向对象的方法来处理。这个好方法的主旨是使每个对象自己学会将自己的状态读出并写入磁盘文件，而不是编写一个函数去记录每个对象的状态和所有的全局变量。

为保存游戏，你所要做的就是编写全局变量然后创建一个简单的函数。由函数通知游戏中的每个对象将其自身的状态写出。然后，将该游戏重新加载，剩下你要做的就是将这些全局变量读入系统、将所有对象的状态装入游戏。

因此，如果你要装入另一个对象或对象类型，加载/保存处理只局限于该对象自身，而不会涉及整个环境。

实现多人游戏

下一个游戏编程的花招是多人游戏编程。当然，如果你想编写一个联网游戏，那就另当别论了——尽管 DirectPlay 使得通信部分变得更为容易。然而，如果你要做的是让两个或两个以上的玩游戏者在同一时间或轮流玩你的游戏，那你只需增加一些额外的数据结构、稍微作一下程序的调整即可。

轮流

轮流的实现既简单又复杂。说其简单是因为既然你能够实现一个玩游戏者，那两个或更多只需多一个或一个以上的游戏玩家的记录即可。说它难是因为在切换时你必须为每一个玩游戏者提供游戏保存的功能。所以通常而言，如果你的游戏需要具备轮流切换选项，你就必须在游戏中实现保存的功能。显而易见，游戏玩家在轮换的时候并不知道游戏已被保存。

为加强印象，下面列出了两人轮流玩的游戏所需的制作步骤：

1. 开始游戏，玩家 1 开始。
2. 玩家 1 玩游戏直到结束。
3. 玩家 1 的游戏状态被保存，玩家 2 开始。
4. 玩家 2 玩游戏直到结束。
5. 玩家 2 的状态被保存（这时进行轮换）。
6. 将玩家 1 先前被保存的游戏重新加载，玩家 1 继续。
7. 返回步骤 2。

你可以看到，轮换发生于步骤 5，随后游戏便在两个玩家间轮流进行。假如游戏的玩家是两个以上，只需简单地让他们间轮流进行（一次只能一个人玩），直到轮到最后一个，然后再从头开始。

多画面设置

实现两个或两个以上的玩家同时在一个屏幕上玩游戏比玩家轮流交换要复杂一些。因为你不得不将游戏编写得复杂一些——将玩家间的游戏规则、冲突和交互考虑进去。而且在同一时刻多人玩的情况下，你必须为每个玩家分配指定的输入设备。这通常是指一个玩家分配一个游戏操纵杆，或一个玩家使用键盘而另一个使用游戏杆。

同一时刻多人参与的游戏还有一个问题是一些游戏并不适于这样做。例如在滚动游戏中，一个玩家想走这条路而另一个玩家却想走另一条路。这将造成抵触，你不得不予以考虑。因此最适于多人同时玩的是单屏幕格斗游戏或多人为了同一个目标而走到一起的游戏。

但如果你允许玩家自由地走动，这时你就必须创建如图 11.14 所示的多画面设置。

多画面设置的惟一问题是多画面显示！你必须产生出两个或更多的游戏画面。这在技术上极具挑战性。此外，屏幕上或许没有足够的空间用于显示多画面，因而玩家很难看到所发生的事情。但最起码它是一个非常酷的游戏选项，不用时你可以将其关闭。



图 11.14 多画面游戏演示

多线程编程技术

到目前为止，本书所谈及的所有演示程序都是使用单线程时间循环和编程模式。时间循环对玩家的输入作出响应并以每秒 30 帧的速度对游戏进行渲染。在对玩家作出反应的同时，游戏每秒要执行几百万次的运算，同时处理几十或上百个小任务，如绘制所有的目标实体、取回输入数据、制作音乐等。图 11.15 示意了这类游戏的标准循环过程。

由图 11.15 可知，游戏逻辑以串行方式来完成各项任务。当然也有例外，比如通过中断来完成一些简单的逻辑任务，诸如音乐、输入控制等。但在极大程度上，此类游戏是一

个长长的由函数调用构成的串行结构。

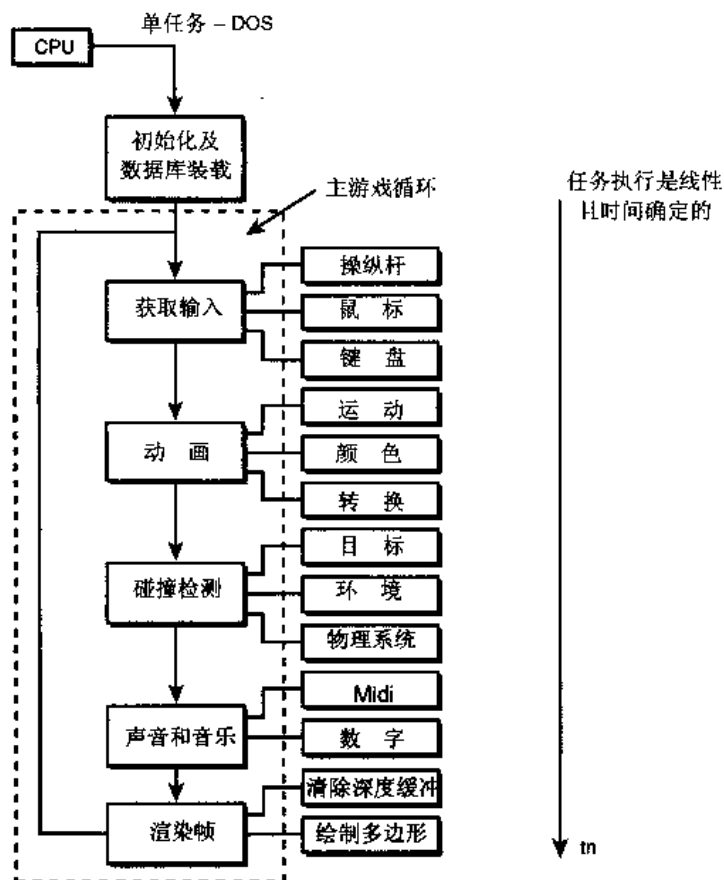


图 11.15 标准 DOS 单任务游戏循环

尽管每一项任务都是顺序执行，但只要机器足够快，其执行的结果就如同同时发生一般，这样便使游戏看上去非常流畅和真实。所以，大多数游戏程序是一个单任务执行线程——许多操作串行执行并输出每一帧所需要的结果。这是解决问题最好的方法之一，也是 DOS 游戏编程的必然结果。

然而在今天，DOS 时代已成为过去，所以现在是发挥 Windows 95/98/2000 多线程能力的时候了！你必须习惯并喜欢这种编程模式！

这部分将就 Windows 95/98/2000 下的多线程执行问题进行探讨。这些线程可以戏剧性地在一个应用程序中完成多个任务的操作。在开始之前，让我们先看一看一些术语。这些术语对你来说并不陌生。

多线程编程的术语

在计算机词典里有许多以“multi-”开头而含义不同的词语。让我们先谈一谈多处理器和多重处理，最后再讨论多线程。

一个多处理器计算机是指具有多于一个处理器的计算机。Cray 和 Connection Machine 就属于这类机器。Connection Machine 计算机可以安装多达 64000 个处理器（构成一个超立体网络），并且每一个处理器均可执行代码。

退而言之，你可以购买到装有四个 PIII+ 处理器、运行 Windows NT 的计算机。一般而言，这些均是 SMP（对称多处理）系统，即四个处理器可以对称地执行任务。实际上，情况并非总是如此，因为操作系统内核可以说是只运行在一个处理器上，但随着任务处理的进行，这些任务便对称地运行在每个处理器上。所以说，多处理器计算是一种应用多处理器来分担负载的计算机系统。

对于某些操作系统，在每个处理器上只能执行一项任务或一个进程，而在其他操作系统上，如 Windows NT，每个处理器上可运行数千个任务。这便是基本的多任务处理——多个任务运行在单个或多个处理器系统上。

最新的概念是多线程，也是今天最令人感兴趣的术语。在 Windows 95/98/2000 下，一个进程不管其是否能够单独地运行，通常情况下，它就是一个完整的程序。它有自己的内存空间、前后关系并能独自退出。

而一个线程则是一个较为简单的程序实体。线程由进程创建，彼此间各不相同、结构简单并运行在进程所在的空间。线程的美妙之处在于能够得到尽可能多的处理器时间，并可以从创建它们的父进程所在的空间退出。

这意味着于线程的通信非常简单。本质上，这恰是游戏程序员所需要的：一个执行线程的任务和其他的主程序任务可以并行地完成工作、访问程序中的变量。

既然带有“Multi-”前缀，因此你有必要搞清楚几个概念。首先，Windows 95、98 和 2000 都是多任务/抢先多任务操作系统。这表明任何任务、进程或线程都不能完全控制计算机。每一项任务、进程或线程在某种程度上都具有抢先性、或被冻结而下一个执行线程开始运行。这与 Windows 3.1 完全不同——在 Windows 3.1 中各项任务不具有抢先性。如果在每个循环中没有调用 GetMessage（...），其他进程就不工作。而在 Windows 95/98/2000 下，你可以设置一个无限循环，而其他任务仍然照常运行。

而且，在 Windows 95/98/2000 下，每一个进程或线程都有一个优先级，这个优先级决定了每个进程或线程在抢先运行前的等待时间。所以，如果有 10 个相同的优先级进程，那么它们的等待时间相同或以循环的方式被处理。可是如果有一个线程具有内核级的优先级，该线程在每个循环中将获得更多的运行时间，如图 11.16 所示。

最后，问题出现了：Windows 95/98/2000/NT 的多线程间有什么区别？它们之间当然有一些区别。但运行于 Windows 95 上的绝大多数程序均可安全地运行在以上所有的平台上。这是最基础的操作系统。尽管 98 和 NT 的稳定性更好一些，但在这部分我仍然使用 Windows 95 机器来运行大部分的程序实例。

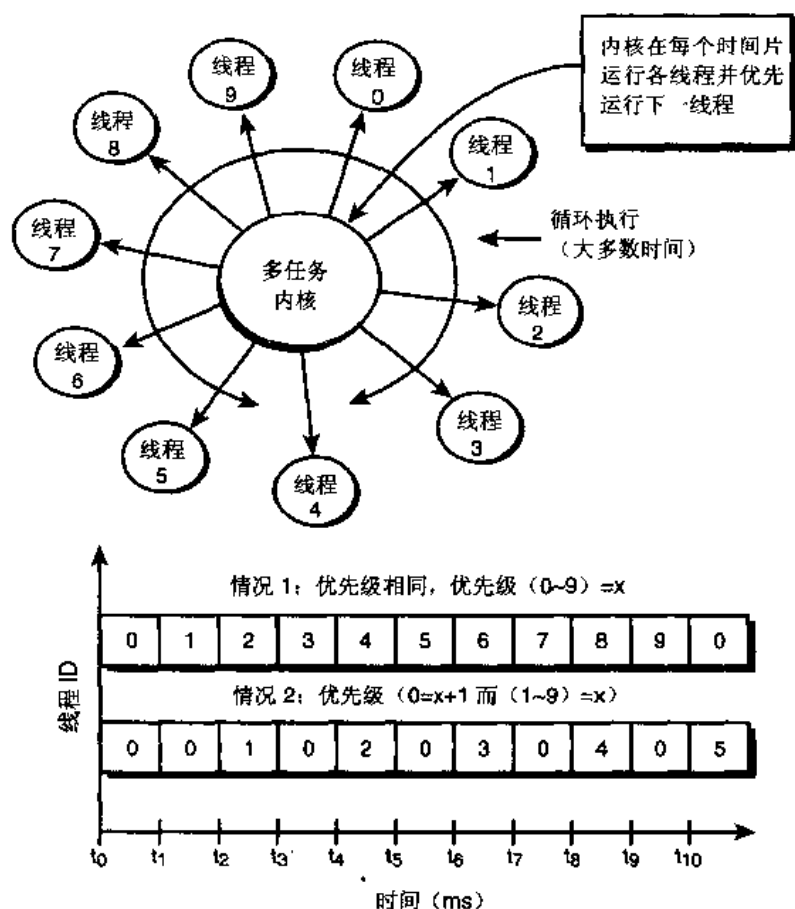


图 11.16 具有相等或不等优先级的循环线程执行过程

为何要在游戏中使用线程？

现在这个答案是显而易见的。事实上，我想你可能创建一个包含 1000 个任务的列表。这时你就可以立刻使用线程来处理。然而，假如你对此仍然一知半解，看看下面一些用到多线程编程的地方：

- 更新动画
- 产生环绕音响效果
- 控制小对象目标
- 查询输入设备
- 更新全局数据结构
- 创建弹出菜单和控件

上述最后一项应用，我也经常使用。在游戏运行时创建菜单并允许玩家改变设置一直是令人头疼的事。但是用线程处理就简单多了。

到目前为止，我依然没有回答为什么要在游戏编程中使用线程而不使用一个庞大循环

和函数调用这个问题。基本上，线程可以完成同样的工作，并且当你所创建的面向对象的程序越来越多，达到一定程度时，你就需要提出类似于自动机的结构。这些便是代表游戏角色的对象——你希望在创建和销毁时对游戏主循环没有逻辑副作用。这可以通过 C++ 类并结合多线程编程来实现。

在开始你的第一个多线程程序前，让我们搞清楚一下事实：在单处理器机器上，一次只能运行一个线程。所以你的选择余地不大，但从软件的观点来看它不失为一种好的方法，因此能够使你的编程更容易、不易出错。图 11.17 示意了一个主线程和三个二级线程同时执行的情况。

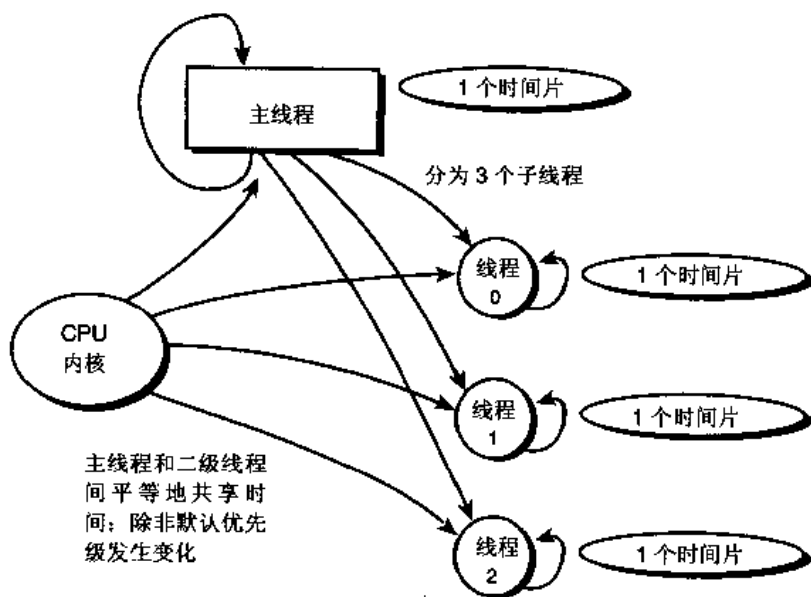


图 11.17 主线程及三个二级线程的示意图

图中的时间表表明了不同的线程对处理器的占用时间，以毫秒计算。如你所见，一次只能运行一个线程，但它们可以按照顺序依次运行并根据优先级的大小来确定运行时间。准备工作足够了！让我们看一些代码吧！

取得一个线程

在下面的例子中，你将使用一个控制台程序。再次强调，请正确地编译这个程序。（我已负担太重，因为我每小时要收到 30~60 封有关错误使用 VC++ 编译器的电子邮件。这些邮件均来自我写的几本书的读者。难道他们没有读前言吗？）

还有一条告诫是：对于这些例子，你必须使用多线程库。进入 MS DEV Studio 的主菜单，在 Build 下拉菜单里有 Setting、CodeGeneratrion 选项，将库设置为多线程，如图 11.18 所示。此外，确保将优化选项关闭。因为有时候该选项会将多线程同步代码搞乱，所以最好将其关掉。

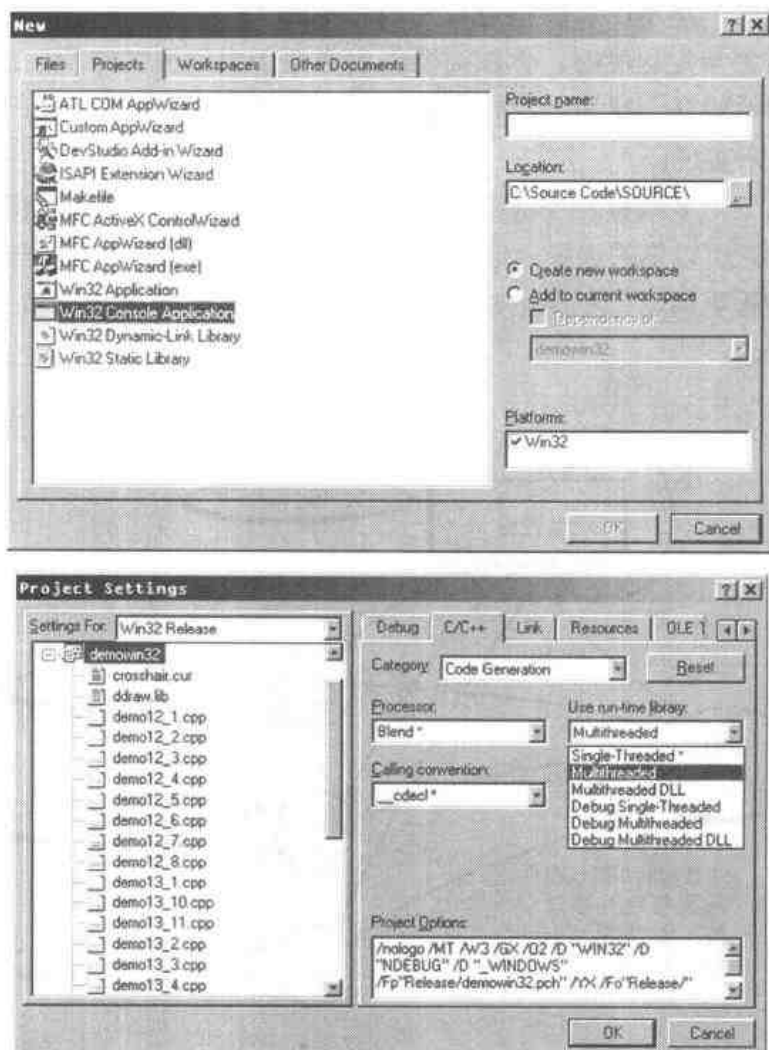


图 11.18 使用多线程库创建一个控制台应用程序

注意

我有一种似曾相识的感觉。真的是似曾相识吗？还是伪装的假象？如果你不能确定你就不能断定它是不是你需要的，不是吗？

一切就绪，让我们开始吧。创建一个线程很简单，而防止其被损坏才是困难的部分！Win32 API 调用的形式如下：

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // pointer to thread security attributes
    DWORD dwStackSize, // initial thread stack size, in bytes
    LPTHREAD_START_ROUTINE lpStartAddress,
    // pointer to thread function
    LPVOID lpParameter, // argument for new thread
```

```
DWORD dwCreationFlags, // create flags
LPDWORD lpThreadId) // pointer to returned thread identifier
```

lpThreadAttributes 指向一个 **SECURITY_ATTRIBUTES** 结构，该结构指定了这个线程的安全属性。如果这个 **lpThreadAttributes** 值为 **NULL**，该线程就以默认的安全描述符创建，并且由此所致的句柄不会继承。

dwStackSize 按字节指定线程堆栈的大小。如果指定值为 0，堆栈的大小就与进程的主线程相同。堆栈在进程的内存空间内自动分配并在进程结束时释放。如有必要堆栈大小可以增加。

CreateThread 调拨由 **dwStackSize** 指定的字节的数量，并在内存不足时返回分配失败消息。

lpStartAddress 指向线程所要执行的、由应用程序提供的函数并代表线程的开始地址。函数接受一个 32 位的参数并返回一个 32 位的退出值。

lpParameter 定义一个传递线程的 32 位参数值。

dwCreationFlags 指定一个附加标志来控制线程的创建。如果 **CREATE_SUSPENDED** 标志被定义，线程就以挂起状态创建而并不执行直到 **ResumeThread()** 函数被调用。如果改制为 0，线程在创建后立即执行。

lpThreadId 指向一个接受线程标识符的 32 位参数。

如果函数执行完毕，其返回值就作为一个句柄传递给下一个新线程。如果函数执行失败，将返回 **NULL**。若要获得更多的错误信息，调用 **GetLastError()** 函数。

函数调用看上去有些复杂，但并非如此。它只是提供了更多的控制功能。大多数情况下，你会用到的功能并不多。

当处理完一个线程时，你应当关闭该线程的句柄。换言之，就是让系统知道你不再使用该对象。该功能通过调用函数 **CloseHandle()** 实现，该函数使用 **CreateThread()** 函数返回的句柄，并将内核中指向该线程的形参的计数值减 1。

对于每个线程都需要这么处理。这不会强行结束该线程，只是用于告诉系统，该线程处于非激活状态。该线程可以自己结束或使用函数 **TerminateThread()** 结束，或者在主线程结束时被操作系统结束。这些我们以后会逐一讨论，现在只需知道这是退出多线程程序之前必须的一个清除操作调用。下面是该函数的原型：

```
BOOL CloseHandle(HANDLE hObject); // handle to object to close
```

hObject 确认打开一个对象句柄。如果函数调用成功，将返回 **TRUE**；如果失败，返回 **FALSE**。调用函数 **GetLastError()** 可得到详细的出错信息。此外，**CloseHandle()** 也适于关闭下列对象的句柄：

- 控制台输入或输出
- 事件文件
- 文件映射

- 人工干预
- 命名管道
- 进程处理
- 信号
- 线程

基本说来, `CloseHandle()` 可以关闭指定对象句柄, 缩减对象的句柄数量, 并执行对象存活期检验操作。

警告



一个新线程的句柄在创建时对该线程具有完全的访问权限。如果没有提供权限描述信息, 该句柄可以被任何需要该线程句柄的函数使用。当提供权限设定时, 所有使用该句柄的访问都要进行权限检查。如果检查结果为拒绝, 那么请求的进程将被拒绝使用该句柄访问该线程。

现在来看一看被传递给函数 `CreateThread()` 的、用来创建一个线程的一些代码:

```
DWORD WINAPI My_Thread(LPVOID data)
{
    // _do work

    // return an exit code at end, whatever is appropriate for you app
    return(26);
} // end My_Thread
```

现在你已具备了创建多线程应用程序所需的一切条件。第一个实例将向你展示一个带有主线程的、单线程程序的创建过程。次线程打印出数字 2, 主线程将打印出数字 1。`DEMO11_5.CPP` 包括整个程序。如下所列:

```
// DEMO11_5.CPP - Creates a single thread that prints
// simultaneously while the Primary thread prints.

// INCLUDES //////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure win headers
                             // are included correctly

#include <windows.h> // include the standard windows stuff
#include <windowsx.h> // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>
```



```
// DEFINES ////////////////////////////////////////

// PROTOTYPES ////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data);

// GLOBALS ////////////////////////////////////////

// FUNCTIONS ////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data)
{
    // this thread function simply prints out data
    // 25 times with a slight delay

    for (int index=0; index<25; index++)
    {
        printf("%d ",data); // output a single character
        Sleep(100);         // sleep a little to slow things down
    } // end for index

    // just return the data sent to the thread function

    return((DWORD)data);

} // end Printer_Thread

// MAIN ////////////////////////////////////////

void main(void)
{
    HANDLE thread_handle; // this is the handle to the thread
    DWORD thread_id;      // this is the id of the thread

    // start with a blank line
    printf("\nStarting threads...\n");

    // create the thread, IRL we would check for errors
    thread_handle = CreateThread(NULL,          // default security
                                0,              // default stack
                                Printer_Thread, // use this thread function
                                (LPVOID)1,      // user data sent to thread
                                0,              // creation flags, 0=start now.
                                &thread_id);   // send id back in this var

    // now enter into printing loop, make sure this takes longer than thread,
    // so thread finishes first
    for (int index=0; index<50; index++)
```

```

    {
        printf("2 ");
        Sleep(100);
    } // end for index

// at this point the thread should be dead
CloseHandle(thread_handle);

// end with a blank line
printf("\nAll threads terminated.\n");

} // end main

Sample output:

Starting threads...
2 1 2 1 2 1 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2
2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
All threads terminated.

```

正如你所看到的输出结果，每一个线程只运行很短的一段时间，然后系统便切换至另一个正等待运行的线程。在这种情况下，操作系统只是简单地在主线程和次线程间切换。

现在让我们试着创建一个多线程程序。你只需对 DEMO11_5.CPP 略加修改便可实现该功能。你只需多次调用 `CreateThread()` 函数，每创建一个线程就调用一次，而且每次传递给所创建线程的数据将是每次打印的输出值，这样便可以区分你所创建的线程。DEMO11_6.EXE 便包含了新修改的多线程程序，下面列出以供参考。注意用于储存线程句柄和标识符的数组的用法。

```

// DEMO11_6.CPP - A new version that creates 3
// secondary threads of execution
// INCLUDES //////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain headers
                           // are included correctly

#include <windows.h>           // include the standard windows stuff
#include <windowsx.h>          // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

// DEFINES //////////////////////////////////////

```

```
#define MAX_THREADS 3

// PROTOTYPES //////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data);

// GLOBALS //////////////////////////////////////

// FUNCTIONS //////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data)
{
    // this thread function simply prints out data
    // 25 times with a slight delay
    for (int index=0; index<25; index++)
    {
        printf("%d ",(int)data+1); // output a single character
        Sleep(100);                // sleep a little to slow things down
    } // end for index

    // just return the data sent to the thread function
    return((DWORD)data);

} // end Printer_Thread

// MAIN //////////////////////////////////////

void main(void)
{

    HANDLE thread_handle[MAX_THREADS]; // this holds the
                                        // handles to the threads
    DWORD thread_id[MAX_THREADS];      // this holds the ids of the threads

    // start with a blank line
    printf("\nStarting all threads...\n");

    // create the thread, IRL we would check for errors
    for (int index=0; index<MAX_THREADS; index++)
    {
        thread_handle[index] = CreateThread(NULL, // default security
            0, // default stack
            Printer_Thread, // use this thread function
            (LPVOID)index, // user data sent to thread
            0, // creation flags, 0=start now.
            &thread_id[index]); // send id back in this var
    } // end for index

    // now enter into printing loop, make sure
    // this takes longer than threads,
    // so threads finish first, note that primary thread prints 4
```

```

for (index=0; index<75; index++)
{
    printf("4 ");
    Sleep(100);
} // end for index

// at this point the threads should all be dead, so close handles
for (index=0; index<MAX_THREADS; index++)
    CloseHandle(thread_handle[index]);

// end with a blank line
printf("\nAll threads terminated.\n");

} // end main

```

Sample output:

```

Starting all threads...
4 1 2 3 4 1 2 3 4 1 2 3 1 4 2 3 4 1 2 3 1 4 2 3 4
1 2 3 1 4 2 3 4 1 2 3 1 4 2 3 4 1 2 3 4 1 2 3 4 1
2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2
3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
All threads terminated.

```

哇！是不是很有趣！创建多线程非常容易吧。如果你头脑反应快，就会对此有些厌烦，并会质疑：每次线程调用都要使用同一个函数？这样做还能够工作正常的原因在于所有的变量都在堆栈中创建，而且每一个线程都有自己的堆栈。所以每个线程都能正常工作。如图 11.19 所示。

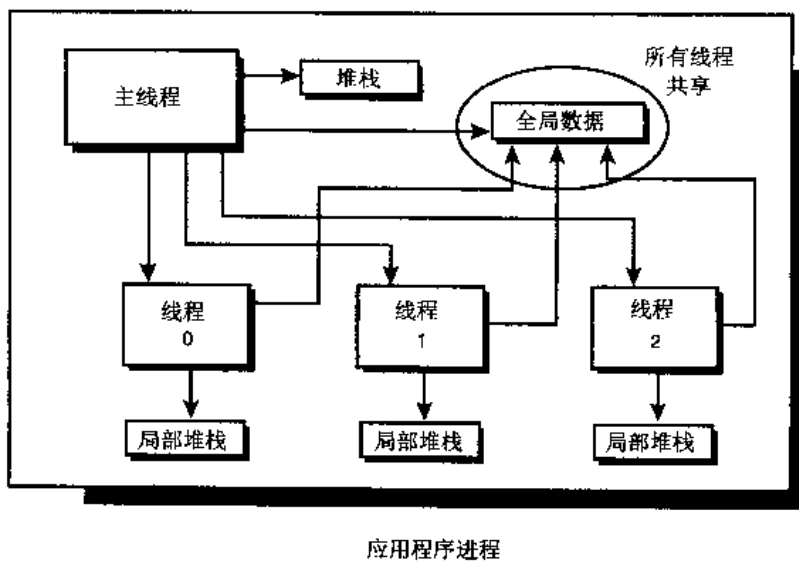


图 11.19 主线程和次线程内存和程序空间分布

图 11.19 忽略了非常重要的一点：结束。两类线程都是自身结束的，但主线程对此没有控制功能。此外，主线程也无法判断其他线程是否已运行完毕或已结束（即它们是否返回）。

我们所要做的是完成线程间的通信和检查各线程的状态。使用函数 `TerminateThread()` 结束函数是一种强制的方法，一般不建议读者使用。

线程间的消息传递

让我们看一看主线程是如何控制其所创建的子线程的。例如，主线程可能需要结束所有的子线程。怎样才能实现呢？有以下两种方法可以结束一个线程：

- 向该线程发送结束信息（正常的方法）。
- 使用内核级的调用并强行结束该线程（不正常的方法）。

尽管在某些情况下需要使用不正常的方法，但这样是不安全的。因为这种方法只是简单地将线程的头指针回收。当该线程需要执行清理操作时，将无法进行。这会造成内存和资源信息的丢失。所以在使用这种方法时要慎之又慎。图 11.20 示意了使用这两种方法结束线程的过程。

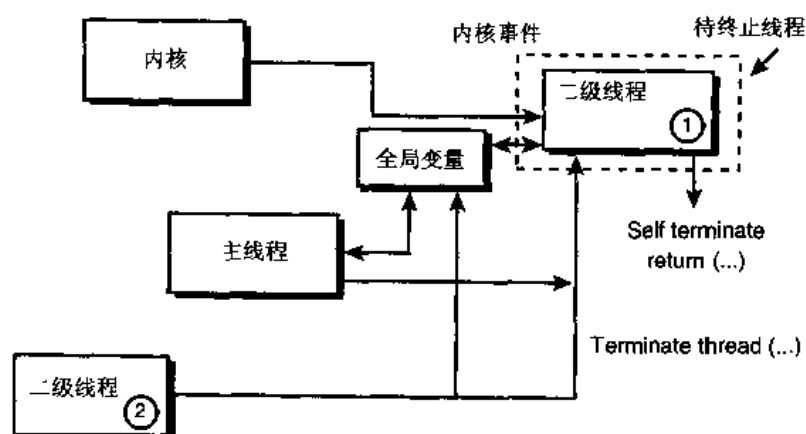


图 11.20 线程终止方法

首先来看一看如何使用 `TerminateThread()` 函数，然后看一个有关向线程发送结束消息以通知该线程执行结束操作的实例。

```

BOOL TerminateThread(HANDLE hThread, // handle to the thread
DWORD dwExitCode); // exit code for the thread

```

`hThread()` 确认要结束的线程。句柄必须包含 `THREAD_TERMINATE` 方法。

`DwExitCode` 定义线程退出代码。使用 `GetExitCode()` 获得线程退出代码。

如果函数调用成功，返回值为 `TRUE`；否则返回值为 `FALSE`。调用函数 `GetLastError()` 可得到详细的出错信息。

TerminateThread()函数用于退出一个线程。当调用该函数时，目标线程就停止执行任何用户代码，并且其初始堆栈不会被释放。而连接到该线程的动态连接库无法知道该线程正在结束，这是调用该函数带来的不良后果。

TerminateThread()函数的用法非常简单，只需简单地调用被结束线程的句柄，并重载返回代码，此后该函数将成为历史。此时不要误解，如果不调用此函数，线程就无法退出。要确保你已经深思熟虑并明白该函数可能带来的后果。

下面介绍由次线程监控的全局变量来实现消息的传递。当次线程检测到该全局变量被设置为结束标志时，次线程便全部结束。可是主线程如何知道所有的次线程在何时结束呢？完成该功能的一个方法是设置另一个全局变量来指示线程数的递减——一个形参类别计数器。

该计数器可以被主线程监测，当它为 0 时说明所有的次线程都已结束，主线程此刻可以安全地继续工作并关闭结束线程的句柄。在阅读一个完整的消息传递系统示例后，我们离真正的编程就不远了。DEMO11_7.CPP/EXE 演示了消息的传递过程，如下所示：

```
// DEMO11_7.CPP - An example of global message passing to control
// termination of threads.

// INCLUDES ////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain headers
                          // are included correctly

#include <windows.h> // include the standard windows stuff
#include <windowsx.h> // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

// DEFINES ////////////////////////////////////////

#define MAX_THREADS 3

// PROTOTYPES ////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data);

// GLOBALS ////////////////////////////////////////
```

```

int terminate_threads = 0; // global message flag to terminate
int active_threads    = 0; // number of active threads

// FUNCTIONS ////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data)
{
    // this thread function simply prints out data until it is told to terminate

    for(;;)
    {
        printf("%d ", (int)data+1); // output a single character
        Sleep(100);                  // sleep a little to slow things down

                                   // test for termination message
        if (terminate_threads)
            break;

        } // end for index

    // decrement number of active threads
    if (active_threads > 0)
        active_threads--;

    // just return the data sent to the thread function
    return((DWORD)data);

} // end Printer_Thread

// MAIN ////////////////////////////////////////

void main(void)
{

    HANDLE thread_handle[MAX_THREADS]; // this holds the
                                       // handles to the threads
    DWORD  thread_id[MAX_THREADS];     // this holds the ids of the threads

    // start with a blank line
    printf("\nStarting Threads...\n");

    // create the thread, IRL we would check for errors
    for (int index=0; index < MAX_THREADS; index++)
    {
        thread_handle[index] = CreateThread(NULL, // default security
                                             0,    // default stack

```

```

        Printer_Thread,    // use this thread function
        (LPVOID)index,    // user data sent to thread
        0,                // creation flags, 0=start now.
        &thread_id[index]); // send id back in this var

    // increment number of active threads
    active_threads++;

} // end for index

// now enter into printing loop, make sure this
// takes longer than threads,
// so threads finish first, note that primary thread prints 4

for (index=0; index<25; index++)
{
    printf("4 ");
    Sleep(100);
} // end for index

// at this point all the threads are still running,
// now if the keyboard is hit
// then a message will be sent to terminate all the
// threads and this thread
// will wait for all of the threads to message in

while(!kbhit());

// get that char
getch();

// set global termination flag
terminate_threads = 1;

// wait for all threads to terminate,
// when all are terminated active_threads==0
while(active_threads);

// at this point the threads should all be dead, so close handles
for (index=0; index < MAX_THREADS; index++)
    CloseHandle(thread_handle[index]);

// end with a blank line
printf("\nAll threads terminated.\n");

} // end main

```


采样输出:

```
Starting Threads...
4 1 2 3 4 2 1 3 4 3 1 2 4 2 1 3 4 3 1 2 4 2 1 3 4 2
3 1 4 2 1 3 4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1
4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1 4 2
3 1 4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1 4 2 3 1 2 3 1 3 2
1 1 2 3 3 2 1 1 2 3 3 2 1 1 2 3 3 2 1 1 2 3 3 2 1 1 2
3 3 2 1 2 3 1 3 2 1 2 3 1 3 2 1 2 3 1 3 2 1 2 3 1 3 2
1 3 1 2 3 2 1 3 1 2 3 2 1
All threads terminated.
```

如输出所示, 当用户敲击一个键时, 所有的线程被结束, 随后主线程也被结束。这种方法有两个问题。第一个问题是错综复杂。下面是它的具体情况, 多看几遍你便会发现这个问题:

1. 假定只剩一个次线程没有关闭。
2. 假定最后一个线程具有处理器控制功能, 并递减跟踪被激活的线程数的全局变量值。
3. 就在这一刹那, 进程切换到主线程。主线程监测全局变量并认为所有的线程都已结束, 然而最后一个线程却还没有返回!

在大多数情况下, 这不是一个问题, 但是如果递减代码和返回代码间有某种联系, 便会出现这个问题。所以我们需要一个函数来询问线程是否已结束。很多情况下, 这是非常有益的。参考一下 `Wait*()` 函数群, 对编程会大有益处。

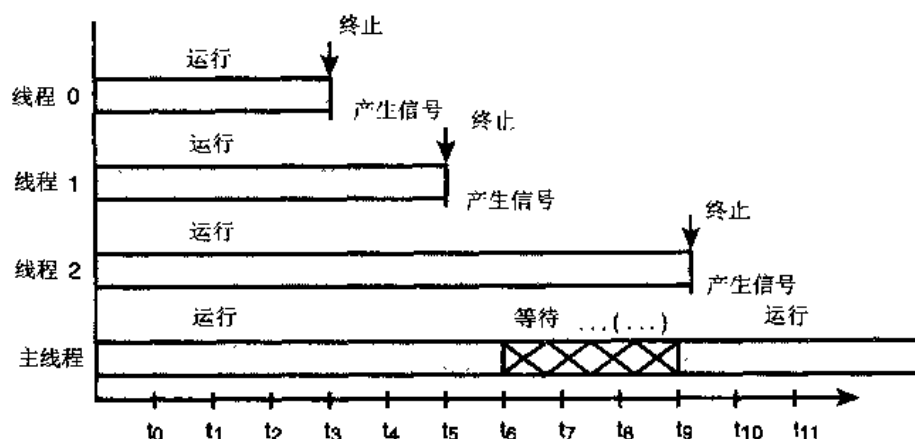
第二个问题是你创建了一个繁忙的循环或轮询的循环。在 Win16/DOS 系统下, 该循环会良好地执行, 但在 Win32 下, 其表现非常拙劣。在一个封闭的循环中, 等待一个变量, 会给多任务内核带来繁重的负担并严重占用 CPU 资源。

可以使用 Windows 附带的 `SYSMON.EXE`、`PERFMON.EXE` 或第三方生产的 CPU 占用率测试工具来进行测定。这些工具软件会有助于你明白线程的运行状态和处理器的占用率。总之, 使用 `Wait*()` 函数群中的函数将有助于你确定一个线程是否结束。

等待最佳时机

我们知道, 任何线程结束时都会产生信号, 而它们运行时就不产生信号。需要关心的是如何监测这些信号。

可以使用 `Wait*()` 函数族来实现消息监测, 该类函数可以实现单信号或多信号的侦测。此外, 你可以调用其中一个等待函数来等待, 直到信号产生。这样做就可以避免繁重的循环。在绝大多数情况下, 这样做要远优于设置一个全局变量的方法。图 11.21 示意了 `Wait*()` 函数的工作机制以及与系统内核、运行程序间的关系。



- t_3 处线程 0 产生信号
- t_5 处线程 1 产生信号
- t_6 处主线程进入等待状态
- t_9 处线程 2 产生信号且主线程被释放

图 11.21 使用 `Wait*()` 函数的信号时间关系

需要使用的两个函数分别是 `WaitForSingleObject()` 和 `WaitForMultipleObject()`。这两个函数分别用于单信号和多信号侦测。它们的定义如下：

```
DWORD WaitForSingleObject(HANDLE hHandle, // handle of object to wait for
    DWORD dwMilliseconds); // time-out interval in milliseconds
```

`hHandle` 用于确定对象。

`dwMilliseconds` 定义超时时间，以毫秒为单位。如果时间间隔已过去，即使侦测对象无信号也要返回。如果 `dwMilliseconds` 值为 0，函数就立刻测读对象的状态并返回。如果其值为无限大，则函数永不超时。

如果函数执行成功，其返回值包含返回的条件状态。如果函数执行失败，调用函数 `GetLastError()` 可以获得详细的出错信息。

函数的返回值类型有以下几种：

- `WAIT_ABANDONED`——所指定的对象是一个互斥对象，该对象在所属线程结束前不能被线程释放。互斥对象的所有权被授予调用线程，互斥对象被设置为无信号。
- `WAIT_OBJECT_0`——指定对象的状态是有信号的。
- `WAIT_TIMEOUT`——超时已过，对象的状态是无信号的。

一般说来，`WaitForSingleObject()` 函数检查指定对象的当前状态。如果对象无信号，调用线程就进入有效的等待状态。在此期间，该线程占用极少的 CPU 时间，直到等待的条件之一得到满足，就结束等待。下面是多信号侦测函数，主要用于多个线程的结束：

```
DWORD WaitForMultiObjects(DWORD nCount, // number of handles
// in handle array
CONST HANDLE *lpHandle, // address of object-handle array
BOOL bWaitAll, // wait flag
DWORD dwMilliseconds); // time-out interval in milliseconds
```

`nCount` 定义 `lpHandle` 所指向的句柄对象数组的数目。对象句柄的最大数目是 `MUXIMUM_WAIT_OBJECTS`。

`lpHandle` 指向句柄对象的数组。该数组包含不同类型对象的句柄。注意对于 Windows NT：句柄必须同步访问。

`bWaitAll` 定义等待类型。如果值为 `TURE`，则 `lpHandle` 数组中所有对象同时有信号时返回。如果值为 `FALSE`，那么任何对象有信号就返回。对于后一种情况，返回值中包含了造成函数返回的状态。

`dwMilliseconds` 定义超时时间（毫秒）。如果超时间隔已过，即使 `bWaitAll` 定义的参数不能得到满足，该函数也要返回。如果 `dwMilliseconds` 的值为 0，函数就立刻侦测指定对象的状态并返回。如果其值为无穷，那么函数永不超时。

如果函数执行成功，其返回值包含返回的条件状态。如果函数执行失败，调用函数 `GetLastError()` 可以获得详细的出错信息。函数返回值主要有以下几种类型：

- `WAIT_OBJECT_0` 到 `(WAIT_OBJECT_0+nCount - 1)` ——如果 `bWaitAll` 值为 `TRUE`，则返回值指出所有指定对象的信号状态。如果其值为 `FALSE`，则返回值减去 `WAIT_OBJECT_0`，这差给出满足等待的对象的 `lpHandle` 数组索引。如果在调用过程中，检测到一个以上的对象有信号时，取信号对象数组索引中的最小值。
- `WAIT_ABANDONED_0` 到 `(WAITa_ABANDONED_0+nCount-1)` ——如果 `bWaitAll` 值为 `TRUE`，则返回值指出所有指定对象的信号状态，而且至少有一个对象是被抛弃的互斥对象。如果其值为 `FALSE`，则返回值减去 `WAIT_ABANDINED_0`，这给出这个满足等待状态的互斥对象的 `lpHandle` 数组索引。
- `WAIT_TIMEOUT` ——超时间隔已过，但不满足由 `lpHandle` 参数指定的条件。

`WaitForMultipleObject()` 函数确定是否满足退出的等待条件。如果等待条件不满足，调用线程就进入一个有效的等待状态，在此状态下，该线程只消耗很少的系统资源。

使用信令来同步线程

这些解释的技术性很强，所以需要举例来说明这些函数的用法。只要对上述例子中的代码稍加修改即可。在下次修改时，我们可以移去全局结束信号标志变量，并创建一个简单调用函数 `WaitForSingleObject()` 的主循环来实现统一功能。

移去全局结束标志变量只是为了使程序变得简单些。它仍然是通知线程结束的最好方法：只是由于处于繁琐的循环中，因此不是监测线程是否已结束的最好方法。

这也是使用 `WaitForSingleObject()` 调用的原因所在。这个调用处于一个占用较少 CPU

时间的、虚拟的等待循环中。而且因为函数 `WaitForSingleObject()` 只能侦测一个信号即只能用于一个线程的结束，所以这个例子中只有一个次线程。

稍后，我们将重写程序，该程序包含三个线程，并使用 `WaitForMultipleObject()` 来结束它们。`DEMO11_8.CPPICPP` 就使用了 `WaitForSingleObject()` 来结束单线程，并创建了另外一个线程，其代码如下：

```
// DEMO11_8.CPP - A single threaded example of
// WaitForSingleObject(...).

// INCLUDES ////////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain
                             // headers are included correctly

#include <windows.h>          // include the standard windows stuff
#include <windowsx.h>         // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

// DEFINES ////////////////////////////////////////////

// PROTOTYPES ////////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data);

// GLOBALS ////////////////////////////////////////////

// FUNCTIONS ////////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data)
{ // this thread function simply prints out data 50
  // times with a slight delay
  for (int index=0; index<50; index++)
  {
    printf("%d ",data); // output a single character
    Sleep(100);         // sleep a little to slow things down
  } // end for index

  // just return the data sent to the thread function
  return((DWORD)data);

} // end Printer_Thread
```

```
// MAIN //////////////////////////////////////////////////////////////////////

void main(void)
{
    HANDLE thread_handle; // this is the handle to the thread
    DWORD thread_id;      // this is the id of the thread

    // start with a blank line
    printf("\nStarting threads...\n");

    // create the thread, IRL we would check for errors
    thread_handle = CreateThread(NULL, // default security
                                0,     // default stack
                                Printer_Thread, // use this thread function
                                (LPVOID)1,      // user data sent to thread
                                0,             // creation flags, 0=start now.
                                &thread_id);    // send id back in this var

    // now enter into printing loop, make sure
    // this is shorter than the thread,
    // so thread finishes last
    for (int index=0; index<25; index++)
    {
        printf("2 ");
        Sleep(100);
    } // end for index

    // note that this print statement may get
    // interspliced with the output of the
    // thread, very key!

    printf("\nWaiting for thread to terminate\n");

    // at this point the secondary thread so still be working,
    // now we will wait for it
    WaitForSingleObject(thread_handle, INFINITE);

    // at this point the thread should be dead
    CloseHandle(thread_handle);

    // end with a blank line
    printf("\nAll threads terminated.\n");

} // end main

Sample output:

Starting threads...
2 1 2 1 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1
```

```

1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1
Waiting for thread to terminate
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
All threads terminated.

```

这个程序很简单。通常在创建次线程后就进入打印循环。当其结束时，调用函数 `WaitForSingleObject()`。如果主线程还有许多工作要做，则继续将其完成。但在本例中，主线程没有其他任务，因而直接进入等待状态。可以使用程序 `SYSMON.EXE` 来察看一下 CPU 的使用状态。在进入等待状态后，可以看到与使用繁重的循环相反，CPU 被占用得极少。

在这里，函数 `WaitForSingleObject()` 有一个使用技巧。假如想知道一个线程在结束前的状态，可以通过 `WaitForSingleObject()` 函数的 `NULL` 调用来实现，源代码如下所示：

```

// ... code
DWORD state = WaitForSingleObject(thread_handle, 0); // get the status

// test the status
if (state == WAIT_OBJECT_0){// thread is signaled, i.e. terminated}
else
    if (state == WAIT_OBJECT_TIMEOUT){// thread is still running}

// ... code

```

简单之至，这是检测一个特定的线程是否已结束的绝妙方法。这一方法结合全局终止变量是一种非常有效的终止线程的方法。同时该方法也是无须进入等待状态而用于检测某个线程在实时循环中是否已终止的好方法。

等待多个对象

我们已经提及了这个问题。`Wait*()` 族函数的最后一个函数就是一个用于终止多线程的函数。我们现在来编写使用该函数的程序。我们所要做的就是创建一个线程数组，然后将句柄的数组与若干形参一起传递给 `WaitForMultipleObjects()` 函数。

当该函数返回时，如果一切正常，那么所有的线程都已终止。`DEMO11_9.CPPIEXE` 与 `DEMO11_8.CPPIEXE` 相似，只不过一个是单线程一个是多线程而已。再次提醒：不要使用全局终止标志，因为我们已知道如何实现这一功能。每个次线程只运行几个循环就结束返回。`DEMO11_9.CPP` 的源代码如下所示：

```

// DEMO11_9.CPP -An example use of
// WaitForMultipleObjects(...)

// INCLUDES ////////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain headers
// are included correctly

```

```

#include <windows.h>           // include the standard windows stuff
#include <windowsx.h>          // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

// DEFINES ////////////////////////////////////////

#define MAX_THREADS 3

// PROTOTYPES ////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data);

// GLOBALS ////////////////////////////////////////

// FUNCTIONS ////////////////////////////////////////

DWORD WINAPI Printer_Thread(LPVOID data)
{
    // this thread function simply prints out data 50
    // times with a slight delay
    for (int index=0; index<50; index++)
    {
        printf("%d ",(int)data+1); // output a single character
        Sleep(100);                // sleep a little to slow things down
    } // end for index

    // just return the data sent to the thread function
    return((DWORD)data);

} // end Printer_Thread

// MAIN ////////////////////////////////////////

void main(void)
{
    HANDLE thread_handle[MAX_THREADS]; // this holds the
                                     // handles to the threads
    DWORD thread_id[MAX_THREADS];      // this holds the ids of the threads

    // start with a blank line
    printf("\nStarting all threads...\n");

```

```
// create the thread, IRL we would check for errors
for (int index=0; index<MAX_THREADS; index++)
{
    thread_handle[index] = CreateThread(NULL, // default security
        0, // default stack
        Printer_Thread, // use this thread function
        (LPVOID)index, // user data sent to thread
        0, // creation flags, 0=start now.
        &thread_id[index]); // send id back in this var
} // end for index

// now enter into printing loop,
// make sure this takes less time than the threads
// so it finishes first
for (index=0; index<25; index++)
{
    printf("4 ");
    Sleep(100);
} // end for index

// now wait for all the threads to signal termination
WaitForMultipleObjects(MAX_THREADS, // number of threads to wait for
    thread_handle, // handles to threads
    TRUE, // wait for all?
    INFINITE); // time to wait, INFINITE = forever

// at this point the threads should all be dead, so close handles
for (index=0; index<MAX_THREADS; index++)
    CloseHandle(thread_handle[index]);

// end with a blank line
printf("\nAll threads terminated.\n");

} // end main
```

采样输出:

```
Starting all threads...
4 1 2 3 4 1 2 3 1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3
1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3
1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3
1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3 1 4 2 3 2 4 1 3
1 4 2 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1
3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1
3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3 2 1 3
All threads terminated.
```

输出正是你所希望的。所有线程和主线程一样都进行了打印输出工作，但当主线程的

循环完成时，次线程将继续进行，直到它们全部完成为止。由于函数 `WaitForMultipleObjects()` 的阻塞作用，所有线程结束后，主线程才终止返回。

多线程编程和 DirectX

现在我们对多线程编程有了一定的了解。下一个问题便是如何将其用于游戏程序和 DirectX 编程。只要去做——这里有你需要的一切。当然，必须确保编译时使用多线程库。此外，在处理大量的 DirectX 资源时，我们还会碰到许多关键的问题。

对于资源要有一个全局规划，以防止一个以上线程访问同一个资源时出现程序崩溃。比如，假定一个线程锁定了一个界面，而另一个线程运行并试图锁定同一个界面。这样就会引起问题。这类问题可以使用 `semaphores`、`mutexes` 和 `critical section` 来解决。这里没时间逐一详细探讨。但读者可以参考相应的资料，如由 Addison Wesley 出版的、Jim Beveridge 和 Robert Weiner 合著的《*Multithreading Applications in Win32*》。这是一本有关这方面内容最好的参考书。

为实现这类资源管理程序并能够正确地共享线程，我们需要创建一个变量来跟踪使用资源的线程。任何线程需要使用可能其他线程正在使用的资源之前都必须检测该变量。当然，除非这个变量能够被检测或自动改变，否则这依然是个问题。因为你可能在改变一个变量过程中而其他线程恰好获得控制权。

可以将这类变量设置为 `volatile` 类型以将其占用资源最小化，这样便告知编译器不要为其进行内存拷贝。然而，最终你还不得不使用 `semaphores`（一个简单的类似于全局变量的计数器，但却是在汇编中以自动代码形式存在，这样做具有不能中断的特性）、`mutexes`（只允许一个线程访问关键代码，是二进制的 `semaphores`）、`critical sections`（指定编译器在编译时一次只能允许一个 Win32 调用）等等。另一方面，如果每一个线程的功能相对比较独立，也就不必对此太关注。

DEMO11_10.CPP1EXE 是一个应用多线程编程的 DirectX 实例，读者可以检验一下该程序。该程序创建了许多外星物体并使它们绕着主线转动。此外该程序还创建了另外一个线程以动态改变这些外星物体的颜色。这是一个非常简单、安全的多线程程序。使用另一个线程来使其颜色动态化不会出现很多问题。注意要确保该程序连接到 `DirectX.LIB` 文件。

但是，如果有许多线程都调用同一函数，就会产生再访问的问题。需要再访问的函数必须要有状态信息，而且不能使用全局变量来获得该信息，因为这可能会被具有抢占优先权的线程进出程序时破坏。

此外，如果使用线程来动画 DirectX 对象，外观、计时及同步过程会严重破坏代码。因此建议读者严格限制使用线程来处理那些在很大程度上是独立的、只存在于它们自己的“状态空间”中的、并且不需要高精度运行的对象。

高级多线程编程

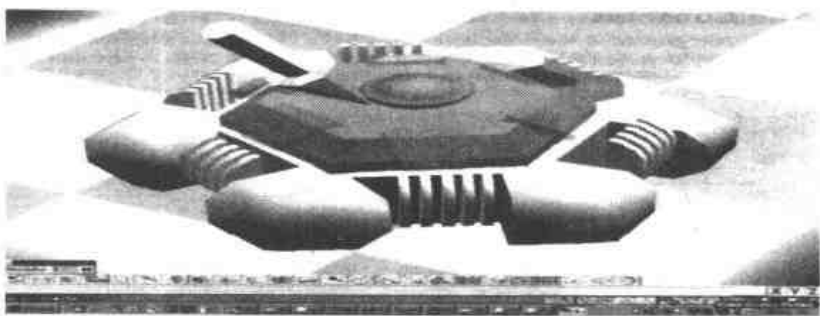
好了！本章到此也告一段落。下一章将主要探讨比赛的条件、死锁、关键部分、互斥体、信号以及实际会碰到的头疼问题。澄清所有这些问题（除了最后一个）都会有助于读者编写多线程程序而不会出问题。当然，即使对此一无所知，读者只要具备一般的常识并明了任何线程都可能会破坏其他线程这个道理，也能够编写出安全的多线程程序。读者应多加注意，线程是访问共享代码的机制。

尽可能使程序的每部分都自动化，确保一个线程不能修改变量以防另一个线程使用该变量时出错。而且，除本章提及的函数调用外，还有几个基本的函数，如 `ExitThread()` 和 `GetThreadExitCode()` 等，但这几个函数相对比较简单易于理解，可以在 API 参考书中查到它们。

总结

本章内容读来比较轻松，没有太多的技术术语，只是一道探讨变量的知识大餐。OK！我想读者们已充足了电。严格说来，在本章中我们共同探讨了许多基础知识：数据结构、内存管理、递归、分析、定点运算和多线程编程。

有些东西与游戏似乎相关不大，但确实相关。要制作一个游戏，读者必须了解编程的每一部分——这非常复杂！在下一章中我们将探讨人工智能这个课题。



12

人工智能在游戏中的运用

本章将回答许多关于人工智能的问题。实际上，人工智能不是仿真，而是基于逻辑、数学、概率、记忆的各种智能的集合——这些我们都有吗？

学习了本章内容之后，就能够编写出比较完善的程序和算法，这些程序和算法可以使游戏中的物体以合理的方式运行，而且可以做任何你想要它们做的事。本章主要内容有：

- ➔ 人工智能入门
- ➔ 单一确定的算法
- ➔ 模式和脚本
- ➔ 行为状态系统
- ➔ 存储和学习
- ➔ 计划和决策树
- ➔ 导航
- ➔ 先进的脚本语言
- ➔ 人工神经网络
- ➔ 遗传算法
- ➔ 模糊逻辑

人工智能入门

人工智能这个词的理论意思是可以使计算机思考或处理信息方式类似于人类的硬件和软件。

人工智能的应用虽然是从几年前才开始的，但现在，人工智能和其他相关领域（比如人工生命、智能元）的研究和应用以指数速率发展。

现在，系统是以有生命的形式存在，任何人都可以定义生命。一些公司已经在虚拟的计算机领域中创造了人工生物，这些人工生物也会生、死、生病、进化、繁殖、悲伤、饥饿等等。

这些技术可以通过人工神经网络、遗传算法、模糊逻辑实现。神经网络近似于人脑；遗传算法是一套技术和推测，运用于基于生物模式的软件系统；模糊逻辑则把理论建立在可靠的推论上。

听起来是不是有点脱离实际？是的。但这些都是真的，而且发展速度很快。记住，克隆以前也只是幻想，而现在却是科学事实。

回到现实中来，我们并不准备在游戏中创建一个与第一流的 AI（人工智能）同样复杂的物体。而是将要看看游戏编程者用来创建智能生物（至少看上去是智能的）的最简单、最基础的技术。实际上，许多游戏编程者仍然对人工智能一窍不通，而且还没有意识到 AI（人工智能）在这个领域的有用性。我觉得 AI（人工智能）及相关的技术对游戏界的影响程度可以与几年前的 DOOM 图像技术相媲美。

实际上，3D 图形已经开始衰退。虽然 3D 图形看起来比较真实，但图形运行起来却很缓慢。我们需要的一流游戏不但视觉效果要很好，而且更重要的是游戏要具有很高的智商，就如我们中的佼佼者一样。

最后，当你读到后面几页并采用上面的代码编写程序时，记住所有的技术只是技术而已。这没有所谓正确方法和不正确方法，只有可以运行工作的方法。如果编写出来的坦克可以达到你要求，这就可以了，如果不能达到，那你需要改进程序和相应的技术。

不管基础的人工智能技术有多么的粗糙，玩家总是把自身的品性映射到游戏中的虚拟对象上。这就是诀窍——玩家相信游戏中的对象真的可以进行策划、决策、思考，就像人类一样。

明确的 AI 算法

明确的算法是预定和既定程序的行为。例如，如果你看看第八章“矢量光栅化及 2D 变换”中介绍的多边形行星演示中的 AI（如图 12.1 所示），这就很简单了。

用 AI 创建了一个行星，使它的运动方向和速率都是随机的。这种类型的智能可以用如下方式表示：

```
asteroid_x+=asteroid_x_velocity;  
asteroid_y+=asteroid_y_velocity;
```

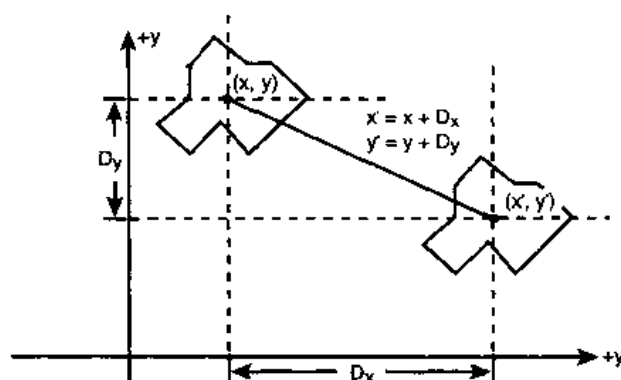


图 12.1 行星 AI

行星有个目标：沿它们的轨迹运动。这个人工智能很简单——这个行星不处理任何输入，也不改变自己的运行路线。从某种意义上说，这个行星是智能的，但它的智能是很容易确定的，而且还可以预测到。这是我介绍的第一种智能类型——简单、可预测、可编程。这种类型的 AI 产生于 Pong/Pac-Man 时代。

随机运动

使物体沿直线或曲线运动只有一个步骤，就是随机地移动物体或改变物体属性，如图 12.2 所示。

例如，你想对一个原子、一只苍蝇或一些没有多大智力但其行为可预测的相似物体建模——它们总以不确定的方式运动。至少看上去是这样。

在建立 AI 模型的最初阶段，你可能想对飞行智能建模，如下：

```
fly_x_velocity = -8 + rand()%16;
fly_y_velocity = -8 + rand()%16;
```

那么，你可以设置物体飞行的周数：

```
int fly_count = 0; // fly new thought counter
// fly in the same direction for 10 ticks of time
while(++fly_count < 10)
{
    fly_x+=fly_x_velocity;
    fly_y+=fly_y_velocity;
} // end while
// .. pick a new direction and loop
```

在这个例子中，飞行的方向和速率是随机的，用这种方式飞行一会，接着采用另外一

种。这听起来像是我在飞行。当然，你可以增加随机性，如改变飞行长度，而不把飞行长度固定为 10 圈。另外，你可以倾向于一个方向。比如，你想让风多往西面吹。

无论如何，可以看出只需要用很少的代码就可以使游戏中的物体看上去比较智能化。参见 CD 中的 DEMO12_1.CPPIEXE。

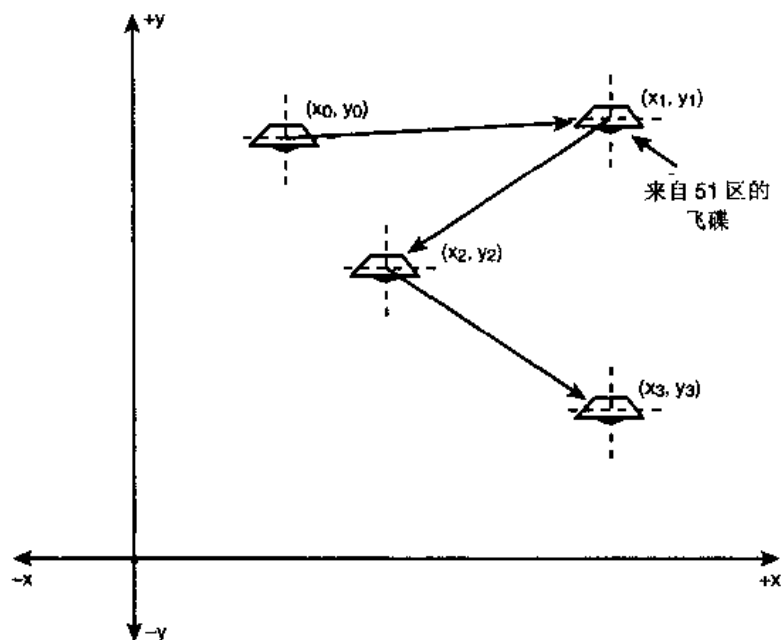


图 12.2 随机运动 AI

跟踪算法

虽然随机运动是不可预测的，但是相当令人厌烦，因为无论怎么随机，都是以同样的方式运动。AI 发展的下一个阶段就是算法，算法把环境中的物体考虑进去，然后反作用于这个物体。我选了一个跟踪算法作为例子。跟踪 AI 把被跟踪物体的轨迹位置考虑进去，然后改变 AI 物体的轨迹使得 AI 物体朝着被跟踪物体的经过的路径运行。

跟踪物体可以直接向物体方向运动，或可以做成更有现实意义的模型，像热追踪导弹一样跟踪物体，如图 12.3 所示。

下面是一个方法比较直接的例子，看看下面这个算法：

```
// given; player is at player_x, palyer_y
// and game creature is at
// monster_x,monster_y
// first test x-axis
if (player_x>monster_x)
monster_x++;
if (player_x<monster_x)
```

```

monster_x--;
// now y -axis
if (player_y>monster_y)
monster_y++;
if (player_y<monster_y)
monster_y--;

```

这段代码简单而实用，Pac-Man 的 AI 也是用相同的方法编写的。当然，Pac-Man 只能直角转动，运动路线也只是直线并回避障碍物。但都差不多。例子参见 CD 中的 DEMO12_2.CPP\EXE。在演示中你可以通过上下键来控制一个幽灵，避免被物体击中。

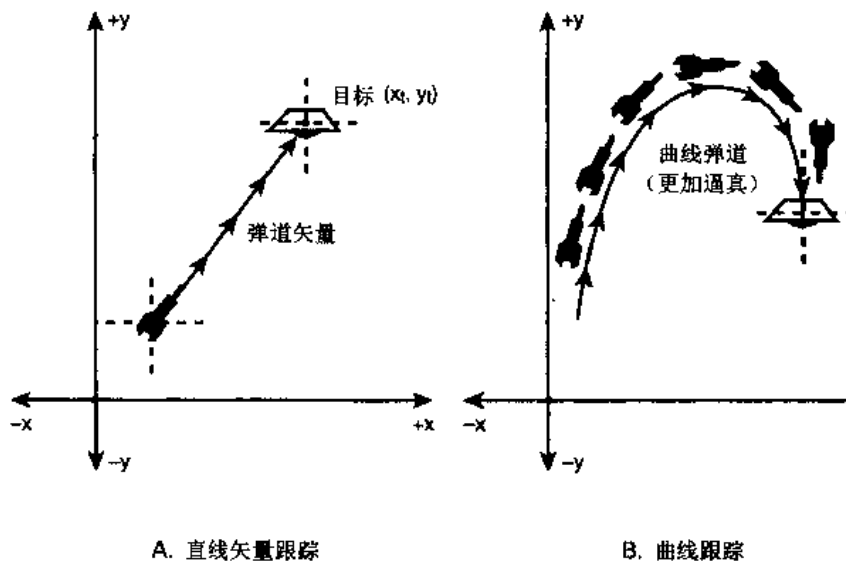


图 12.3 跟踪方法

这种跟踪形式很好，但看起来太人工化了。因为 AI 控制的物体能够精确地跟踪对象。跟踪物体一个更自然的方法是根据跟踪物体中心和被跟踪物体中心，改变跟踪物体的轨迹矢量的方向。如图 12.4 所示。

这个算法工作条件：假设 AI 控制物体称为 **tracker**（跟踪物体），并有如下属性：

```

Position: (tracker.x,tracker.y)
Velocity:(tracker.xv,tracker.yv)

```

被跟踪物体称为 **target**（目标），并有如下属性：

```

Position: (target.x,target.y)
Velocity:(target.xv,target.yv)

```

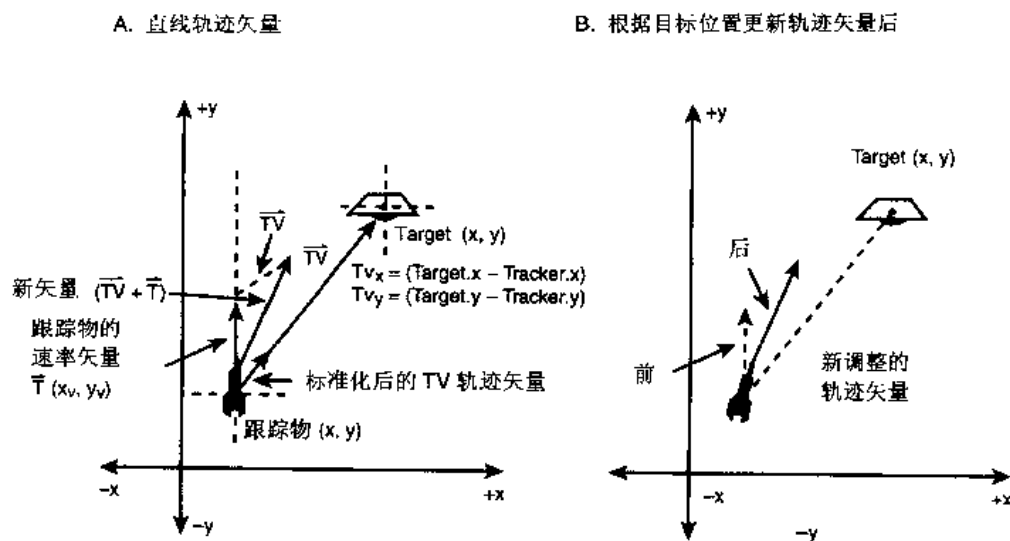


图 12.4 根据轨迹矢量跟踪物体

基于这些定义，下面是调整跟踪物体的速率矢量的一般逻辑步骤：

1. 计算从跟踪物体到目标的矢量：

$TV = (target.x - tracker.x, target.y - tracker.y) = (tvx, tvy)$ ，使 TV 标准化——换句话说，就是用 (tvx, tvy) 除以 $VECTOR_LENGTH(tvx, tvy)$ ，所以最大长度是 1.0，把这称为 TV^* 。

注意 $VECTOR_LENGTH()$ 只是计算从原点 $(0,0)$ 引出的矢量的长度，也就是 $\sqrt{x^2 + y^2}$ 。

2. 通过加上 TV^* （按照一个比率估计），调整跟踪物体的现行速率矢量：

```
tracker.x += rate * tvx;
tracker.y += rate * tvy;
```

注意到当比率大于 1.0 时，跟踪引导汇集更快，跟踪算法跟踪目标更加接近，改变目标的运动更加迅速。

3. 改变跟踪速率矢量后，速率矢量可能会溢出。换句话说，一旦锁定，跟踪物体就会继续在目标的方向上加速。所以，你应该对速率设定一个上限，速率达到上限后上使跟踪物体减速。下面是例子：

```
// get magnitude of velocity vector
tspeed = Vector_Lengh(tracker.xv, tracker.yv)
//moving too fast?
If (tspeed > MAX_SPEED)
{
    // shrink the velocity vector
    tracker.xv *= 0.75;
    tracker.yv *= 0.75;
}
```



```

        tracker.yv*=0.75;
    } // end if

```

还有其他的选择——0.5 或 0.9——任意。如果目标很完整，甚至可以计算出确定的溢出量，然后按照这个数量减小矢量。我知道现在我们还没有接触到矢量数学，而在这个例子中用到了这个术语，所以我要给出一个例子，这个例子中有一些使用这个算法的跟踪代码。我们来看看这个例子怎样执行前面的步骤：

```

// mine tracking algorithm
//compute vector toward player
float vx = player_x - mines[index] .varsI[INDEX_WORLD_X];
float vy = player_y - mines[index] .varsI[INDEX_WORLD_Y];
//normalize vector (sorta :)
float length = Fast_Distance_2D(vx , vy);
// only track if reasonable close
if (length<WIN_MINE_TRACKING_DIST)
{
    vx=MINE_TRACKING_RATE*vx/length;
    vy=MINE_TRACKING_RATE*vy/length;
    //add velocity vector to current velocity
    mines[index].vx+=vx;
    mines[index].yv+=vy;
    //add a little noise
    if ((rand()%10)==1
{
    vx=RAND_RANGE(-1,1);
    vy= RAND_RANGE(-1,1);
} //end if
//test velocity vector of mines
length=Fast_Distance_2D(mines[index].xv,mines[index].yv);
//test for velocity overflow and slow
if (length >MAX_MINE_VELOCITY)
{
    // slow down
    mines[index].xv*=0.75;
    mines[index].yv*=0.75;
} // end if
} // end if
else
{
    vx=RAND_RANGE(-2,2);
    vy=RAND_RANGE(-2,2);
    // add velocity vector to current velocity
    mines[index].xv+=vx;
    mines[index].yv+=vy;

    //test velocity vector of mines
    length=Fast_Distance_2D(mines[index].xv, mines[index].yv);

```

```

//test for velocity overflow and slow
if (length>MAX_MINE_VELOCITY)
{
    // slow down
    mines[index].xv*=0.75;
    mines[index].yv*=0.75;
} // end if
} // end if
} // end else

```

虽然，这些代码很明显是从一个程序中的一个循环或处理水雷的数量的代码中得到的，但没关系。这段程序有些地方值得注意。比如，里面有一段代码测试水雷是否与玩家保持在一定距离之内。如果不是，水雷就不能跟踪玩家，但在它的轨迹运行时伴随着的一些随机的声音有小小的改变。另外，甚至当水雷跟踪到玩家时，我加了一些随机声音。这些声音可以使得跟踪看起来更有真实感。在空间中、水中、空气中或其他地方，重力、密度等均会有一些轻微的改变。因此，加点声音上去会使物体看起来更加真实。

轨迹跟踪算法的例子可参见 CD 中的 DEMO12_3.CPPIEXE。在这个演示中你可以在一个滚动的领域中来回移动一艘小船。通过上面的算法，在这个领域中有水雷跟踪你。演示中的控制键为：

箭头键	控制船
Ctrl 键	发射船上的武器
+/- 键	改变跟踪速率
H 键	切换显示
S 键	切换扫描

注意怎样降低跟踪速率使得跟踪物体看起来很有把握追上物体。这是一个很好的示范，所以有很多地方值得学习。把它学好。

提 示



由于我要用 GDI 绘制文本，所以正文显示使得游戏运行非常慢。我想让你知道这一点。在真正的游戏中，你应该用自己的字体引擎绘制文本。

反跟踪：逃脱算法

下一个 AI 技术就是使游戏中的物体能够逃脱跟踪。想想 Pac-Man 中的幽灵是怎样逃脱的。用 AI 创建一个逃脱很简单。实际上，你已经有了逃脱算法的代码。前面的跟踪代码与你想要的逃脱算法恰好相反，只要把代码作一些小小改动，你就会得到逃脱算法。下面就是改动后的代码：

```

// given: player is at player_x, player_y

```

```

// and game creature is at
// monster_x,monster_y
// first test x-axis
if (player_x < monster_x)
    monster_x++;
if (player_x > monster_x)
    monster_x--;
// now y -axis
if (player_y < monster_y)
    monster_y++;
if (player_y > monster_y)
    monster_y--;

```

注 意

你应该注意到上面代码中没有等于(==)。这是因为在这个例子中，我不想让物体运动。我只想让物体成为玩家的一员。如果你想让物体运动，你可以用上==，让这个例子做些其他的工作。

现在你可以只用随机运动、追踪、逃脱，就能够创建出一个给人印象相当深刻的 AI 系统。实际上，你有足够能力做出一个 Pac-Man 的脑子来！运行 CD 中 DEMO12_4.CPP\EXE，检验运动中的逃脱。这和 DEMO12_2.CPP 相似，但这是逃脱人工智能而不是跟踪人工智能。下面我要介绍模式。

模式和基础控制脚本

算法和确定性算法很重要，但有时你需要使游戏中的物体按步骤或按不同的脚本运动。比如：当你发动汽车时，你要执行下列步骤：

1. 从口袋中掏出车钥匙。
2. 把钥匙插入锁中。
3. 打开车门。
4. 进入车中。
5. 关车门。
6. 把钥匙插入到点火装置中。
7. 转动钥匙。
8. 发动车。

上面这些步骤你可能没有想过，但你每次发动汽车时都要重复这些步骤。当然，如果有些事发生了，你可能要改变这些步骤的顺序。模式是智能行为重要的一部分，人，或这个星球上的智能生命的物体都用到模式。

基本模式

为游戏中的物体创建一个模式的难易程度取决于游戏物体本身。比如，可以很容易创建一个运动控制模式。例如你要编写一个类似于 Phoenix 或 Galaxian 的枪战游戏，外来的攻击者必须遵循从左到右的模式，然后在某一点上用个特定的模式攻击你。这种模式或脚本化的 AI 可以通过不同的技术实现，但我认为基于解释运动指示的技术是最简单的。如图 12.5 所示。

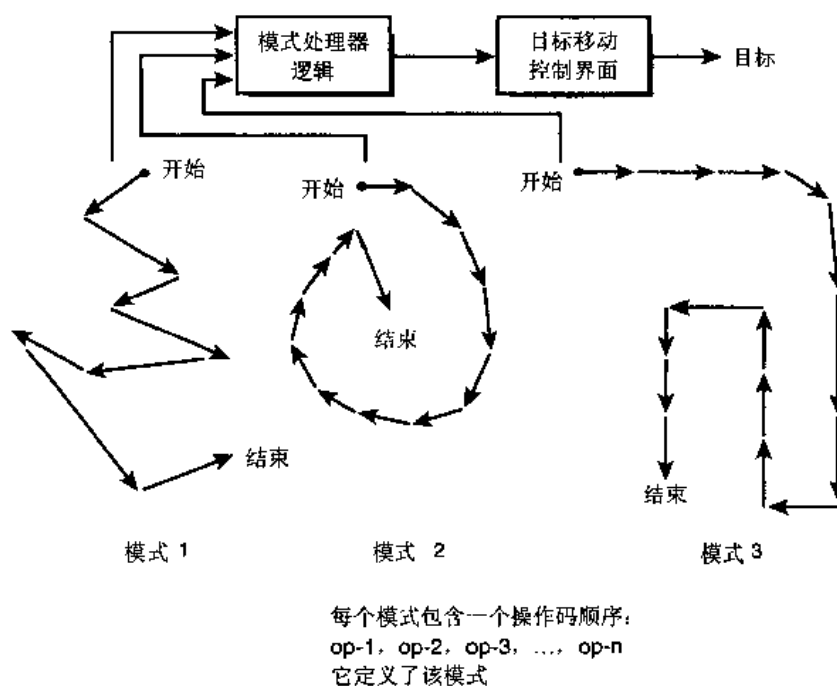


图 12.5 模式引擎

每个运动模式存储为指令或命令的序列号，如表 12.1 所示。

表 12.1 命令语言模式的设置值

命 令	值
GO_FORWARD	1
GO_BACKWARD	2
TURN_RIGHT_90	3
TURN_LEFT_90	4
SELECT_RANDOM_DIRECTION	5
STOP	6

每个直接的命令可能都有另一个操作数或进一步限制命令的数据（比如操作时间）。结果，语言命令格式模式看起来与下面的表达式很相似：

INSTRUCTION OPERAND

INSTRUCTION 是从前面的表中得到的（通常编码为一个单数字），OPERAND 是可以进一步限制命令行为的数值。用这个简单的命令格式，编写一个程序（命令的顺序）来定义模式。然后为原模式写一个解释程序，就可以适当地控制游戏物体。

例如，模式语言格式化后，第一个数字是命令本身，而第二个数字表示运动的时间。用旋转和停止创建一个方形模式将会很繁琐，如图 12.6 所示。

下面是一个编码为[INSTRUCTION, OPERAND]格式的例子：

```
int num_instructions = 6; // number of instruction in script pattern
// this holds the actual pattern script
int square_stop_spin{
    1,30,4,1, // go forward then turn right
    1,30,4,1, // go forward and turn right
    1,30,4,1 // go forward and turn right
    1,30,    // go forward and finish square
    6,60,    // stop for 60 cycles
    4,8,}); //spin for 8 cycles
```

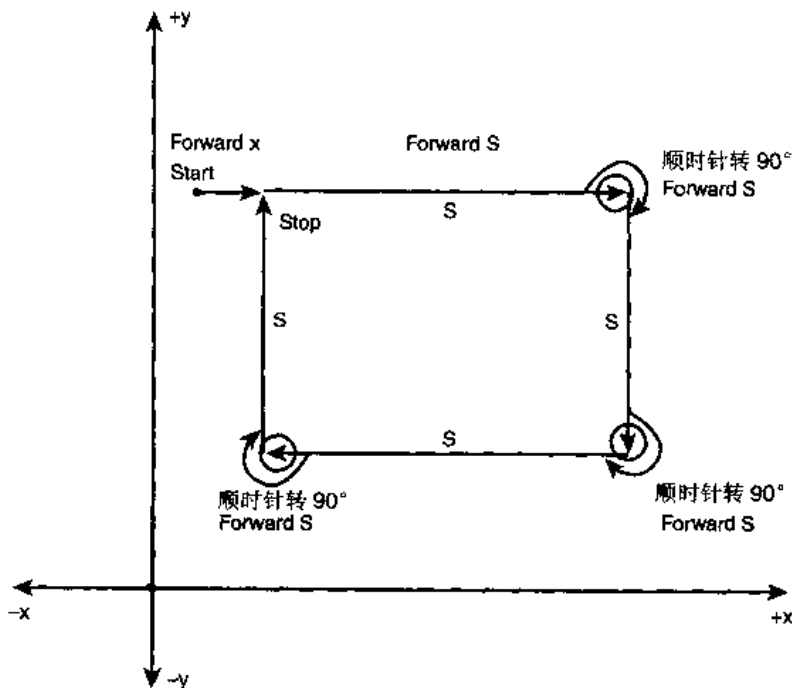


图 12.6 一个具体的方形模式

注意



当然，你可能想用一個比數組更好的數據結構。例如，用使用一個包含記錄列表（在[INSTRUCTION, OPENAND]格式中）和命令個數的類。用這種方法，你可以很容易地創建一個這些結構的數組，每個都包含不同的模式，然後選擇一個模式，放到模式處理器中。

處理模式命令，你所需要做的是編寫一個大的 switch() 函數，函數中解釋了每個命令，並通知遊戲物體將要做什么。如下：

```
// points to first instruction (2 words per instruction)
int instruction_ptr = 0;
// first extract the number of cycles
int cycles= square_stop_spin[instruction_ptr+1];
// now process instruction
switch(square_stop_spin[instruction_ptr])
{
case GO_ORWARD: //move creature forward...
    break;
case GO_BACKWARD: // move creature backward...
    break;
case TURN_RIGHT_90: // turn creature 90 degrees right ...
    break;
case TURN_LEFT_90: //turn creature 90 degrees left...
    break;
case SELECT_RANDOM_DIECTION: // select random dir...
    break;
case STOP:// stop the creature
    break;
} // end switch
// advance instruction pointer (2 words per instruction)
instruction_ptr+=2;
// test if end of sequence has been detected...
if (instruction_ptr > num_instruction*2)
{ /* sequence over */}
```

當然，你應該加上邏輯跟蹤循環計數器，使物體運動。

所有的模式可以匯集成合理運動。因為遊戲物體依賴模式，所以可能決定物體選擇一個使物體和其他物體相撞的模式。如果模式人工智能不把這考慮進去，就會盲目的跟着模式運行。結果，你的模式人工智能中就必須有一個反饋回路來指示人工智能是否非法操作或是運行不可能、不合理邏輯，如果是這樣就重新設置為其他模式。如圖 12.7 所示。

思考一下模式的程序。用這些程序你可以記錄許許多多的運動和戰鬥模式。使用其他人工智能技術在任何合理的时间里幾乎都不可能創建出一個模式，而用一個工具（你將要編寫的）在一分鐘之內就可以創建出一個模式（這記錄在一個文件中），然後在你的遊戲中可以體現出來。用這個技術，可以使得遊戲物體看起來非常靈活。這個技術幾乎用在所有的遊戲中，包括絕大多數戰鬥遊戲，如 Tekken、Soul Blade、Mortal Kombat 等等。

而且，你沒有必要不使用動作模式。你可以用模式來控制武器的選擇，或控制動畫等。

对模式的使用并没有限制。关于运动模式的例子，参见 DEMO12_5.CPPIEXE，上面是一个怪物使用一些模式来回运动，并时常选用新的模式。

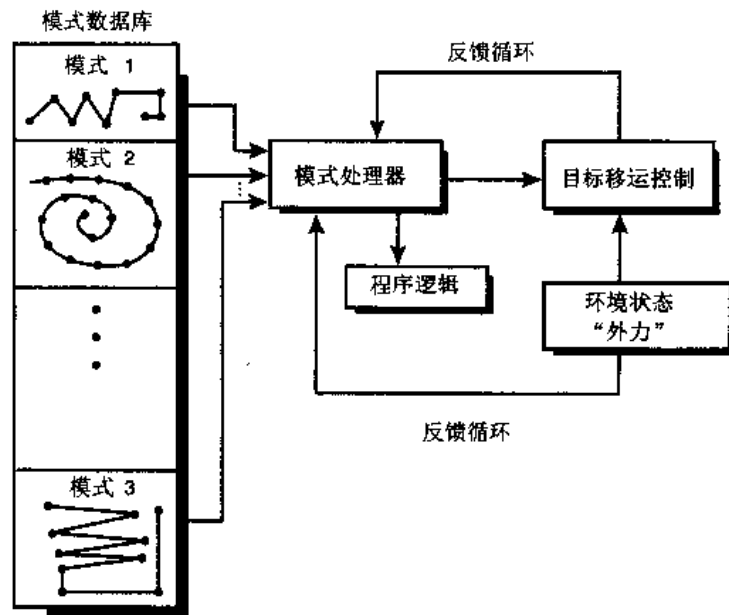


图 12.7 有反馈控制的模式引擎

有逻辑条件的模式

模式使用比较方便，但它具有确定性。也就是说，玩家一旦记住了一个模式，模式就没有用了。玩家总可以打败你的 AI，因为玩家知道下一步将发生什么。这个问题和忽然出现的模式的其他问题的解决方法就是在模式中加上一些逻辑条件，使选择模式不仅仅基于随机选择，还要把游戏环境和玩家的实际情况考虑进去。如图 12.8 所示。

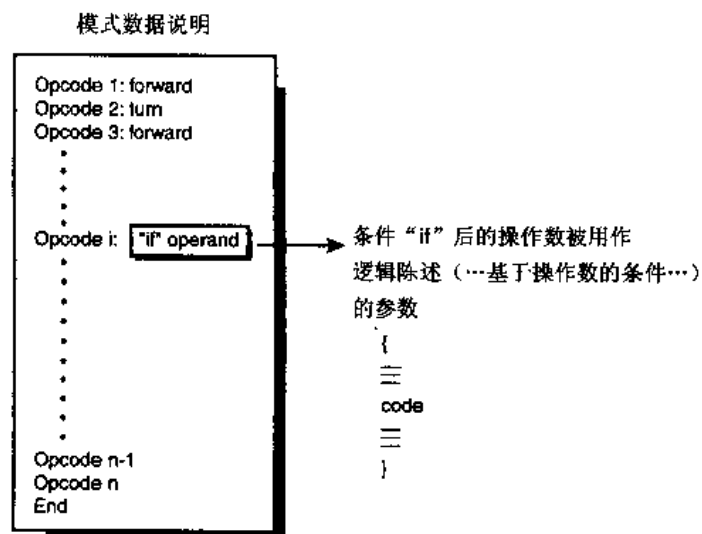


图 12.8 含条件逻辑的模式

有逻辑条件的模式可以用多种等级控制 AI 模型——你可以选择包含条件分支的模式，也可以根据逻辑条件选择模式。例如，你给模式语言添加一个新的命令，这就是一个逻辑条件测试：

```
TEST_DISTANCE 7
```

TEST_DISTANCE 条件可以通过从物体执行模式中测试出物体与玩家的距离进行工作。如果距离太近，太远，或无论哪一种，模式 AI 驱动都有可能改变它所处的模式，这样有助于使物体看起来更为灵活。比如，你键入一个 TEST_DISTANCE 命令到一个标准模式中，这个模式中有很多个命令，如下：

```
TURN_RIGHT_90 , GO_FORWARD, STOP , ...TEST_DISTANCE,
...TURN_LEFT_90, ...TEST_DISTANCE, ...O_BACKWARD
```

物体在一个模式中运行，但每次都有可能会遇到一个 TEST_DISTANCE 命令，在测试玩家位置时，模式 AI 使用操作数跟踪 TEST_DISTANCE 命令。如果与玩家距离变远了，模式 AI 就会停止现行的模式转换到另外一种模式中去。或更好的话，转换到一个确定的跟踪算法以接近玩家。参考下面的代码：

```
if (instruction_stream[instruction_ptr] == TEST_DISTANCE)
{
    // obtain distance, note that on the test
    // instruction the operand is no
    // longer a time or cycle count
    // but becomes context dependent
    int min_distance = instruction_stream[instruction_ptr];
    // if test if player is too far
    if (Distance(player, object) > min_distance)
    {
        // set system state to switch to track
        ai_state = TRACK_PLAYER;
        // .. or you might just switch to
        // another pattern and hope
        // that the object gets closer
    } // end if
} // end if
```

在这，对可以在模式脚本中执行的条件测试的复杂性没有限制。另外，你可能想创建一个飞行中的模式，然后使用。这种例子可以模仿玩家的运动方式。你可以对玩家每次杀死游戏中对象所采取的方式进行采样，然后用同样的策略对待玩家！

这样的技术（虽然不是很完善）用在许多体育游戏中，比如，足球、篮球、曲棍球等许多运动和策略游戏。游戏物体可以做预定的运动，也可以“临时改变主意”。

DEMO12_6.CPPIEXE 中运用了这种条件技术。你在上面可以用箭头键控制物体，屏幕上有一个 AI 骨架。只要你离得不是很远，这个骨架可以任意地选择模式，然后它为了引起你的注意，就会追逐你。

行为状态系统建模

在这，你已经看到一些不同形式出现的限态体——使光线闪烁的编码、主要的时间循环状态机等。现在我想把 FSM（限态体）生成 AI 的过程形式化。

创建一个真正的稳固的 FSM，需要有两个属性：

- 一个合理的状态数量，每一个状态代表一个不同的目标或目的。
- 给 FMS 输入许多信息，例如环境的状况和在环境中的其他物体。

“一个合理的状态数”的前提很容易理解和估计。人类有几百种（或许有几千种）情感状态，每一种状态，还可以进一步分为子状态。问题在于游戏中的角色应该能够以自由的方式运动，这是最起码的。例如，你可能建立下面状态：

状态 1： 向前运动。

状态 2： 向后运动。

状态 3： 转弯。

状态 4： 停止。

状态 5： 发射武器。

状态 6： 追逐玩家。

状态 7： 躲避玩家。

状态 1 到 4 比较简单，但状态 5、6、7 可能需要对子状态建模。这就意味着需要不止一条语句才能得到状态 5、6、7。例如，追踪玩家可能要包括转弯、向前运动。看看图 12.9 描述的子状态概念。然而，不要认为子状态必须基于实际存在的状态，它们可以是所述状态中人工加上的。

这个状态的讨论的关键在于游戏物体需要有足够的变化来做“智能”事情。如果只有两种状态——停止、向前，就没有那么多运动方式。记得那些遥控汽车只能向前运动，然后左转弯。很好笑是吧？

现在来看看稳定的 FMS 的第二个属性，你需要从游戏中的其他物体，从玩家和环境得到反馈或输入。如果你只是简单地输入一个状态然后运行，直到结束，那就相当的没意思。状态已经被灵活地选择出来了，但这是在 100 毫秒之前。现在物体改变了，玩家只须做一些 AI 需要反映的事情。FMS 需要跟踪游戏状态，如果有必要，抢先从先行状态到另一个状态。

如果你把这些都考虑进去，你就可以创建出一个 FMS，这个 FMS 能够模仿一般的经验行为比如攻击、好奇等。我们通过具体的例子来看看这是怎样工作的，从简单的状态机

构开始，然后是先进的个人 FMS。

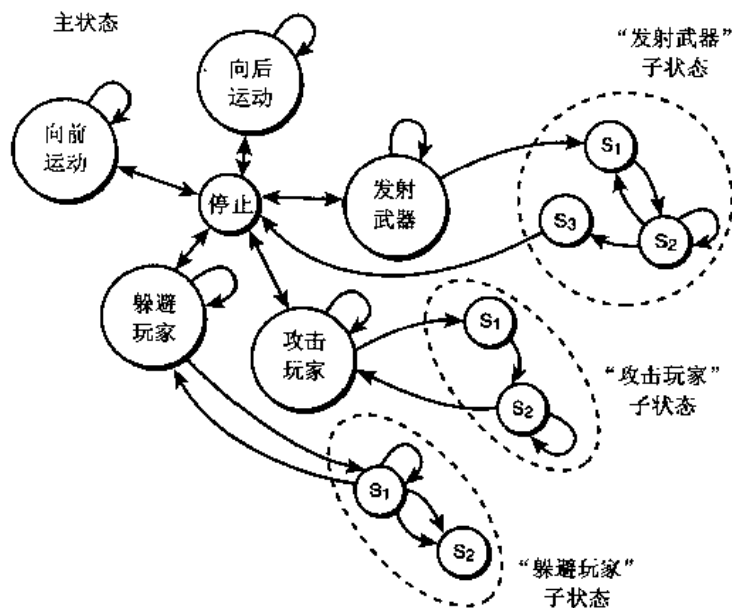


图 12.9 包含子状态的 FMS

基本状态机

这里，你应该看到不同的 AI 技术之间有很多相似之处。例如，模式技术是基于最低级的限态体，这个限态体执行实际的运动或效果。我想做的就是将限态体的等级转换到另一个等级，高等级的状态能够被一些简单的逻辑条件、随机性、模式所利用。本质上，我想创建一个可以指导和命令游戏物体的虚拟的制导系统。

为了更好地了解我所说的，我们用上述的技术对一些行为建模。除了这些行为之外，我们将放置一个标准 FMS 来显示和设置事件和目标的一般指令。

大多数游戏中包含冲突。不管冲突是游戏的主题还是游戏的附属主题，毕竟大多数时间里，玩家是在毁灭敌人或/和炸毁物品。我们要获得一些性能。这些性能是在给出恒定对手的攻击时游戏物体需要存活所要用到的。图 12.10 描述了下面这些状态之间的关系：

- 主状态 1： 攻击。
- 主状态 2： 撤退。
- 主状态 3： 随意移动。
- 主状态 4： 停或中止一会。
- 主状态 5： 寻找物体——食物、能量、光线、黑暗或其他计算机控制物体。
- 主状态 6： 选择模式，并按其运行。

你应该能够看出这些状态与前面例子之间的区别。这些状态在比较高的等级下使用，而且这些状态一定包含子状态或更多的逻辑。例如，可以用确定的算法得到状态 1 和状态

2, 而状态 3、4 也仅仅使用几行代码就能得到。另一方面, 状态 6 则非常复杂, 因为它能命令游戏物体必须执行由主 FSM 控制的复杂模式。

就像你看到的一样, 你的 AI 已经变得相当完善了。状态 5 可能是其他的确定算法, 甚至是确定算法和既定程序查找模式的混合。如果你想自上而下对游戏物体建模, 第一件要想到的是你想使游戏物体的 AI 有多复杂, 然后再利用每个状态和算法。

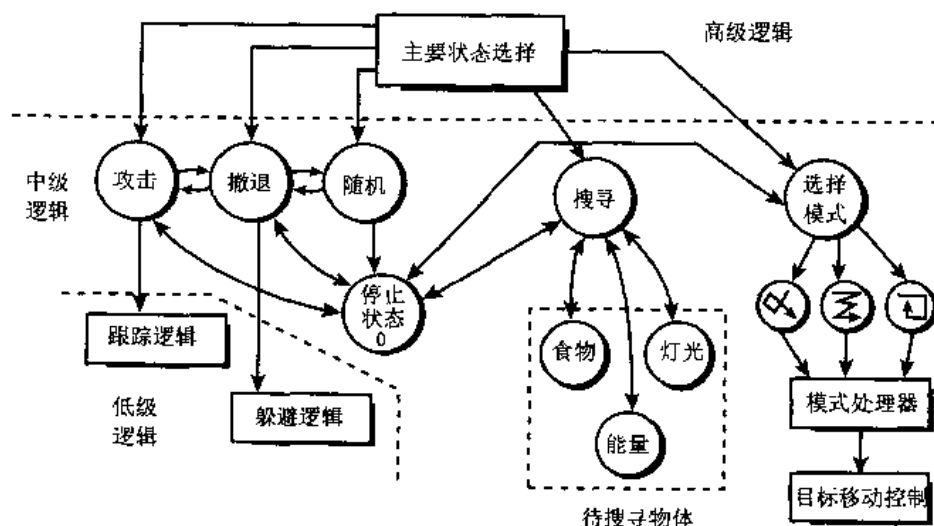


图 12.10 建立一个更好的制导系统

如果你回过头看看图 12.10, 你同样可以看出除了主 FMS (自己选择状态), AI 模型还有其他一部分可以选择状态。这与物体的“想法”和“议程”相似。有很多种方法可以使用这个功能, 比如随机选择、逻辑条件或其他的一些方法。现在, 只要知道在现行的游戏状态下, 必须用智能方式来选择状态就可以了。

下面这一部分代码用了主状态的最初版本。这些代码只能实现一部分功能, 因为完整的 AI 代码需要占用好几页, 但这些代码包含了最重要的结构单元。基本上, 你要填写所有的空白、细节、概括, 然后用到你的程序中。现在, 假设游戏只是由 AI 物体和玩家组成。下面是代码:

```
// these are the master states
#define STATE_ATTACK 0 // attack the player
#define STATE_RETREAT 1 // retreat from player
#define STATE_RANDOM 2 // move randomly
#define STATE_STOP 3 // stop for a moment
#define STATE_SEARCH 4 // search for energy
#define STATE_PATTERN 5 // select a pattern and execute it
// variables for creature
int creature_state = STATE_STOP, // state of creature
    creature_counter = 0, // used to time states
```

```

    creature_x      =320,    //position of creature
    creature_v      =200,
    creature_dx      =0,      // current trajectory
    creature_dy      =0;
// player variables
int player_x = 10;
    player_y =20;
// main logic for creature
// process current state
switch(creature_state)
{
    case STATE_ATTACK;
    {
        // step 1: move toward player
        if (player_x >creature_x) creature_x++;
        if (player_x <creature_x) creature_x--;
        if (player_y >creature_y) creature_y++;
        if (player_y <creature_y) creature_y--;

        // step 2: try and fire cannon 20% probability
        if ((rand()%5) == 1)
            Fire_Cannon();

    } break;

    case STATE_RETREAT:
    {
        // move away from player
        if (player_x >creature_x) creature_x--;
        if (player_x <creature_x) creature_x++;
        if (player_y >creature_y) creature_y--;
        if (player_y <creature_y) creature_y++;
    } break;

    case STATE_RANDOM:
    {
        // move creature in random direction
        // that was set when this state was entered
        creature_x+=creature_dx;
        creature_y+=creature_dy;
    } break;

    case STATE_SEARCH:
    {
        // pick an object to search for such as
        // an energy pellet and then track it similar
        // to the player
        if (energy_x > creature_x) creature_x--;
        if (energy_x < creature_x) creature_x++;
        if (energy_y > creature_y) creature_y--;
        if (energy_y < creature_y) creature_y++;
    }
}

```

```

        } break;
    case STATE_PATTERN:
    {
        //continue processing pattern
        Process_Pattern();
        } break;
    default: break;
    } // end switch
    // update state counter and test if a state transition is
    // in order
    if (--creature_counter <=0)
    {
        // pick a new state, use logic, random, script etc.
        // for now just random
        creature_state = rand()%6;

        // now depending on the state, we might need some
        // setup...
        if (creature_state == STATE_RANDOM)
        {
            // set up random trajectory
            creature_dx = -4+rand()%8;
            creature_dy = -4+rand()%8;
        } // end if
        // perform setups on ther states if needed

        // set time to perform state, use appropriate method...
        // at 30 fps, 1 to 5 seconds for the state
        creature_counter = 30 + 30 * rand()/5;
    } // end if

```

我们来谈谈上面这些代码。首先，现行的状态是经过处理的。这包括逻辑、算法，甚至对其他 AI 进行功能调用，如模式处理。状态执行之后，状态计数器更新代码测试这个状态是否结束。如果结束了，那么就会选择一个新的状态。如果新的状态需要建立，就会执行初始化。最后，用一个随机数选择一个新的状态数，开始新的周期。

你能够做许多改进。可以把状态转换和状态处理混合，也可以在创建状态转换和状态选择时采用更多的逻辑。

添加个性化的更稳定行为

个性只是可预测行为的集合。例如，我的一个朋友有非常“粗暴”的个性。我警告你如果说一些他不喜欢的东西，他很可能让你知道四脚朝天的感觉。而且，他非常急躁，不喜欢想很多。另外，我的另外一个朋友体形非常小，也很懦弱。他知道自己身材，他不敢说出自己的心里话，因为这样很可能会遭到攻击。所以他有非常被动的个性。

当然，人类比例子所描述的复杂得多，但可以对人们有个适当的描述。因此，你应该能够使用逻辑和根据一些行为特点的概率分布建立个性化类型模型，然后设置每个模型的出现概率。这个概率图表可以用来进行状态转换。如图 12.11 所示。

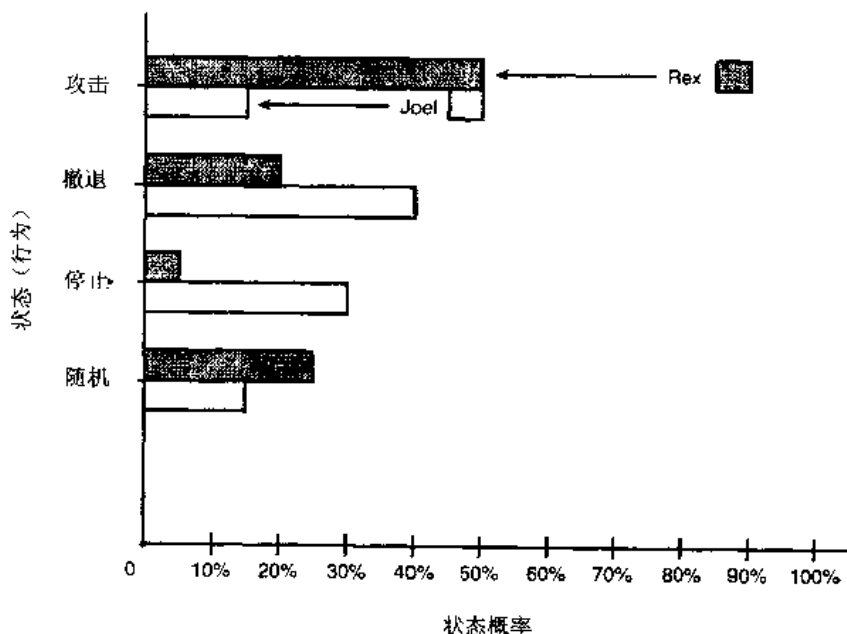


图 12.11 基本行为状态的概率分布

这个模型中有 4 种状态：

- 状态 1： 攻击
- 状态 2： 撤退
- 状态 3： 停止
- 状态 4： 随机

不像前面那样随机地选择一个新的状态，你要建立一个概率分布来把每个物体的个性定义成这些状态的函数。例如，表 12.2 描述了我的朋友 Rex（粗暴）和 Joel（懦弱）的概率分布。

表 12.2 行为概率分布

状 态	Rex (x)	Joel (x)
攻击	50%	15%
撤退	20%	40%
停止	5%	30%
随机	25%	15%

上面的数据是假设的，但看起来却很有意义。Rex 喜欢不加思考地攻击，而 Joel 想得

太多，如果可以，他喜欢跑开。另外，Rex 不是一个有计划的人，所以他做了很多随机事件——撞墙，吃玻璃，欺骗他的女朋友——而 Joel 知道他将要做什么。

这个完整的例子完全是编造的，Rex 和 Joel 根本不存在。但我想你一定把 Rex 和 Joel 的特征记住了，或知道很多人像他们。所以，我的推测是正确的——一个人的外在的行为定义他们的个性为其他人形成的感觉（至少通常的方法是这样的）。这对你的 AI 模型或状态选择非常有用。

用概率分布技术，你只要建立一个有 20~50 个条目（每个条目就是一种状态）的表，然后填写这个表，就可得到你想要的概率。当你选择一个新的状态时，这个状态有一个出现的概率。例如，下面是 Rex 的概率表（概率表为 20 个元素的数组）——这就是说，每个元素有 5% 的可能性：

```
int rex_pers[20] = {1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,3,4,4,4,4,4}
```

除了这个技术，你可能想要加上影响半径。这就意味着要根据变量转换概率分布，比如到玩家或其他物体的距离，如图 12.12 所示。图中显示当游戏物体离得太远时，就会转换到不攻击的寻找模式，而不是近距离所用的攻击战斗模式。换句话说，这时采用了另一个概率表。

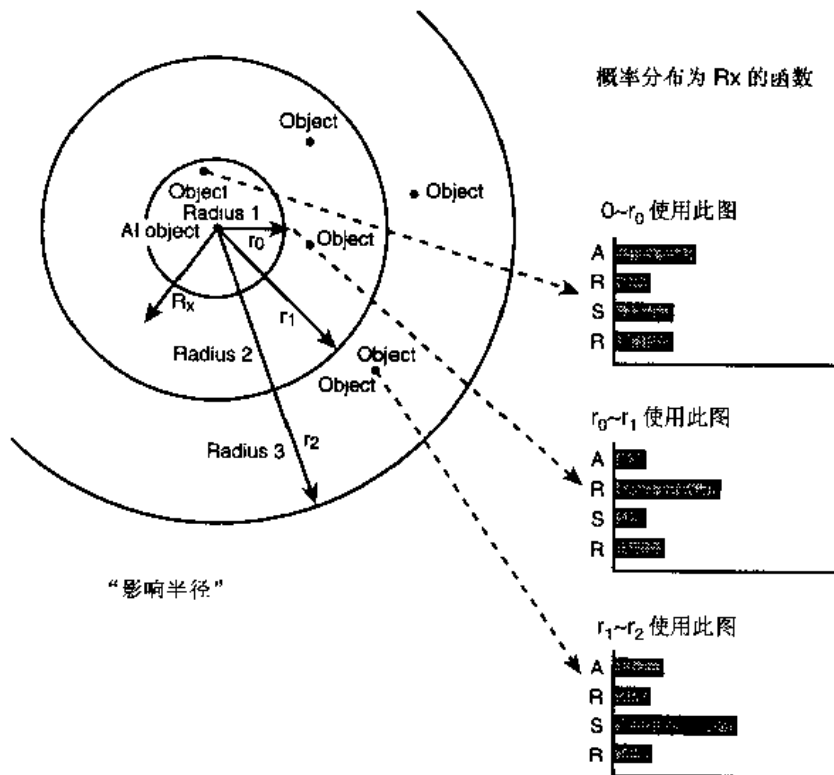


图 12.12 根据距离转换个性概率分布

应用软件对存储和学习建模

一个先进的 AI 有一些其他元素——存储和学习。作为游戏中的 AI 控制物体，这些物体是由状态、条件逻辑、模式、随机数、概率分布等控制。然而，这些物体只是临时思考，不会参考以前的历史记录来作出决定。

例如，如果游戏物体处于攻击模式，玩家不停向右边逃避，物体就会一直落空？你可能想使物体能够跟踪玩家，记住在每次攻击时玩家向右逃避，然后可能改变物体的导向目标作为校正。

另外一个例子是，想像游戏物体自己寻找武器，就像玩家一样。然而，每次物体想得到武器，它必须随机寻找武器（可能用一个模式）。老实说，很容易实现存储，但很少游戏程序员这样做过，因为他们没时间或认为这样做不值得。绝对不是这样的！存储和学习很有用，玩家会注意到。找到能够很容易实现存储和学习的方法是很值得的，而且这对 AI 的决策有很明显的作用。

这些是对存储的一般看法，但怎样运用到游戏中呢？这要根据情况而定，例如，如图 12.13 所示。

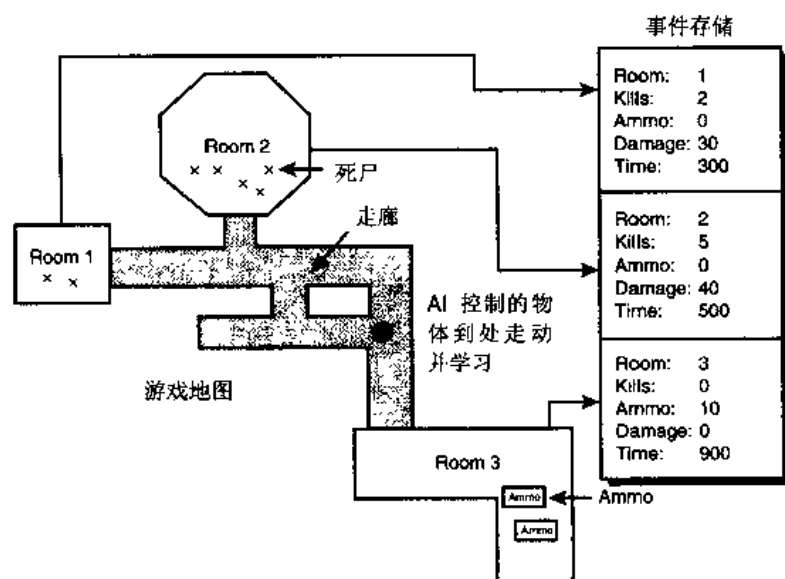


图 12.13 使用暂时地理存储

图是一张游戏地图，上面有每个房间的记录。这些记录存储了下面的信息：

- 被杀死次数
- 玩家对房间的损坏
- 找到的武器

在房间的时间

每次，物体都通过它的 AI 行动，你也想有个更稳定基于存储和学习的选择过程，你要参照事件的记录——房屋的物体存储。例如 当物体进入一个房间时，你可能要检查这个物体是否在房屋里遭受大量的伤害。如果是，物体就会返回，进入另一个房间。

另一个例子，物体可能用完了武器。不是随机地搜寻更多的武器，而是浏览一下它去过的所有房间的存储，看看哪个房间里的武器最多。当然，AI 必须要及时更新存储，但这很容易做到。

另外，你可以使物体之间互相交换信息！例如，如果一个物体在走廊里意外碰到另外一个物体，他们可以合并记录，互相学习对方的游历。或可能强大的物体把力量加载给弱小的物体，因为强大的物体很明显有更好的属性和经历，更容易生存。而且，如果一个物体知道玩家的最后的位置，它能够用信息影响另一个物体的存储，它们就可以集中对付玩家。

怎样使用存储和学习没有限制。最棘手的一部分是把存储和学习比较公正地运用到 AI 中。例如，让游戏 AI 看到整个游戏过程，并存储起来，这是很不合适的。AI 应该像玩家一样必须经过探索才知道具体情况。

提示



许多游戏编程者喜欢运用位串或位向量来存储数据。这更加简洁，而且很容易就交换一位，可以模拟遗忘。

我做了一个蚂蚁模拟作为存储应用的例子，参见 DEMO12_7.CPPIEXE，如图 12.14 所示。

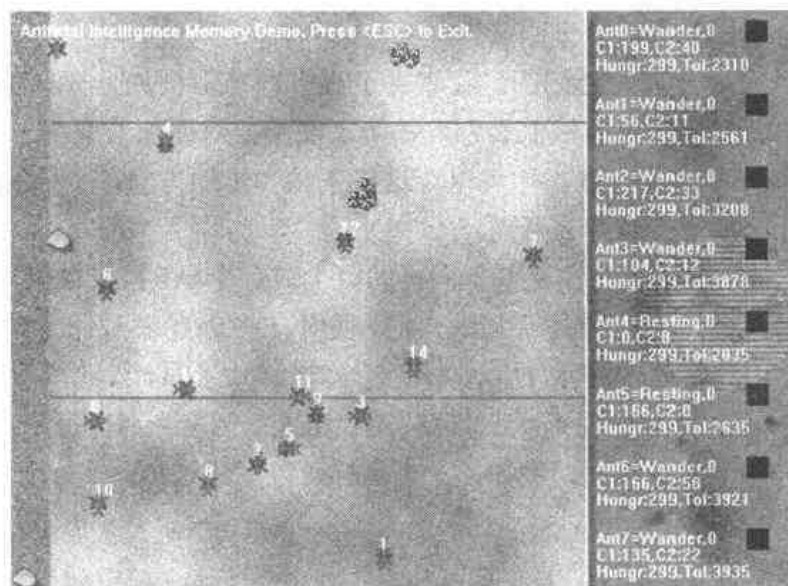


图 12.14 存储演示

模拟从一些红色蚂蚁和几堆绿色食物开始。蚂蚁在找到食物堆之前任意游荡。当蚂蚁找到食物时，它们就开始吃这些食物，直到吃饱。然后又开始任意游荡。当蚂蚁饿了，它们会记得在最近发现食物的地方，然后会朝这个方向运动。

另外，当两个蚂蚁遇见时，它们会交换信息。如果一个蚂蚁没有及时找到食物，它会很可怕地死去。你可以根据系统的处理能力，调整蚂蚁的数量。现在是 16 只，但是只有显示前面 8 个的状态信息和存储画面的空间。这些信息显示在屏幕的右方，信息有当时状态、饥饿程度、饥饿耐受度、两个内部计数器。

你可以加上些东西，使蚂蚁离开废弃物，并建立一个循环系统，在这个系统中食物不会用完。

计划和决策树

到现在为止，所有的 AI 技术都是相当被动和直接——意思是没有使用计划和高水平的逻辑推理。你已经知道怎样使用低水平的 AI，我现在要谈谈高水平的 AI。这通常称为计划编制。

一个计划只是一组高水平的行动，执行这些行动可以到达一个目标。这些行动是有步骤的，它们是以一定的顺序执行来达到目的。另外，在执行任意特定的行为之前必须满足有些条件。例如，下面这些就是看电影的计划：

1. 查找想看的电影。
2. 在电影开始前 30 分钟开车到电影院。
3. 到了电影院，买票。
4. 看电影。电影完了，开车回家。

这看起来是一个很合理的计划，但我省去了很多细节。例如，要在哪里查找想看的电影？你怎样开车？万一身上没有钱怎么办？等等。要不要这些细节取决于计划的复杂程度，但是通常有一些条件和辅助方案，你可以用这些条件和辅助方案作为计划的特定细节，所以对于将要做些什么就不会有什么疑问。

在 AI 游戏中使用计划算法也是基于同样的概念。有个 AI 控制的物体，你想让它根据一些计划行动，并达到一些目的。因此，你必须用一些语言来对计划建模——通常是用 C/C++，但可能要用到特殊的高水平脚本。无论如何，除了对计划建模，你还必须对所有物体（这是计划的一部分）建模：行为、目标、行为条件、目标条件。每一项只是一个 C/C++ 结构或类，并有一些字段在里面。例如，一个目标可以写成这样：

```
typedef struct GOAL_TYP
{
    int class;        // the class of goal
    char *name;       // the name of goal "kill leader"
    int time;         // time until goal expires
}
```

```

int *subgoals // pointer to sub goal list that must be
              // satisfied
int (*eval)(void); //function pointer to determine if
                  // goal has been satisfied
// mor data
} GOAL, *GOAL_PTR;

```

当然，这些定义只是例子，你可能有更多的字段，不过现在你已经知道使用的方法。你必须创建一个能够代表游戏中任何目标的结构，从“炸毁桥”到“寻找食物”。

你需要的下个结构是一个一般的行为结构，这个结构代表为达到某一目标的计划的一部分，物体必须这样做。同样，这得自己动手做，这个结构必须能够反映你想要 AI 做的事情。下面就是一个行为结构：

```

typedef struct ACTION_TYP
{
int class; // class of action
int *name; // name of action
int time; // time allotted to perform action
RESOURCE *resource; // a link to a record that describes
                    // the resources that this action might
                    // need
CONDITIONS *cond; // a link to a record that describes
// all the conditions that must be met
// before this action can be made

UPDATES *update; // a link to a record that describes
                 // all the updates and changes that
                 // should be made when this action is
                 // complete
int (*action_functions)(vid); // a function ptr(s) to an
                              // action function that does
                              // the work of the action
} ACTION, *ACTION_PTR;

```

可以看出，这相当抽象。关键是这些结构在使用中可能完全不同——只要这些结构赋予计划、行为、目标的功能，这就足够了。

编程计划

有几种方法编写计划。你可以把计划编写成实现行为、目标的纯硬件代码，而计划本身是纯 C/C++。这在以前是很普通的技术。一个游戏编程者只要编写执行条件的代码，并设置变量，然后调用函数。这实质上是一个硬件编码计划。

编写计划一个更好的方法是使用形成规则和决策树。形成规则只是一个由一些前提和结果组成的逻辑命题：

```
IF X OP Y THEN Z
```

X 和 Y 是前提，Z 是结果。OP 是任意逻辑运算，如 AND、OR 等。另外，X 或 Y 可能是由其他的形成规则组成；也就是说，它们可能是嵌套的。例如：

```
if (P > 20) AND (damage < 100) THEN consequence
```

或用 C/C++ 表示：

```
if (power > 20) && (damage < 100)
{
    consequence();
} // end if
```

所以形成规则就是一个条件语句。硬件编码计划只是形成语句与行为、目标的集合。编写“计划者”的关键是对这些抽象的东西建模。虽然可以使用硬件编码 C/C++，但创建一个能够读形成规则，包含行为、目标，并代表一个计划的结构更好。

能够帮助你执行这个系统的结构称为决策树。如图 12.15 所示，决策树只是一个树结构，结构中，每个节点代表一个形成规则或/和一个行为。

然而，不是用硬件代码生成这个树，游戏编程者或等级设置者将文件或数据提供给 AI 引擎，而树是由这些文件或数据建立的。这种方法得到的树很普通，不需要重新编译就能工作。现在我们来创建一个小型 AI 计划语言，然后通过一些输入变量和一套行为可以控制一个物体。

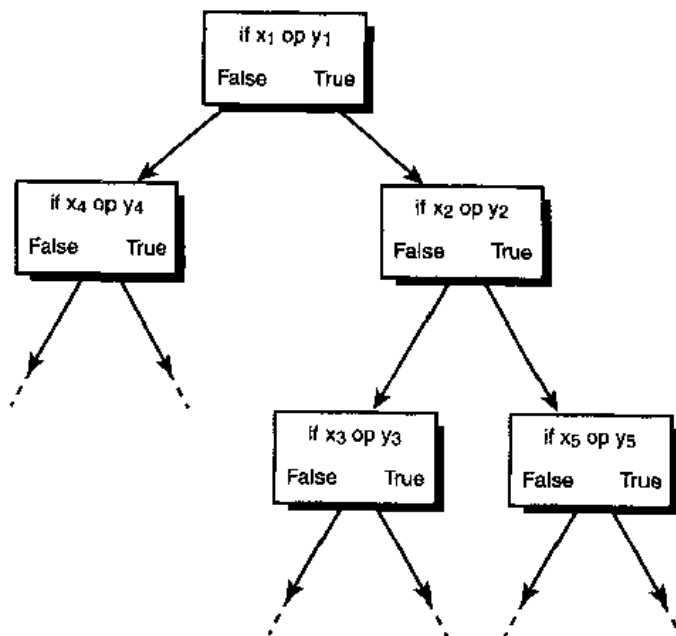


图 12.15 决策树编码产生规则

可以被 AI 物体测定的输入：

DISPLAY	离玩家的距离 (0~1000)
FUEL	剩余的燃料 (0~100)
AMO	剩余的弹药 (0~100),
DAM	当前的损伤 (0~100)
TMR	用虚拟时间表示的现行游戏时间
PLAYST	玩家的状态 (攻击, 没有攻击)

AI 物体可以执行的行为：

FW	向玩家发射武器
SD	自毁
SEARCH	寻找玩家
EVADE	躲避玩家

现在我们开始编辑决策数结构。假设一个或两个前提可以用代码中的 AND、OR 测试，如果你想对它们求反用 NOT。可以使用 >、<、=、或 != 比较前提本身和输入（或一个定量）。另外，任何代码中有一个 TRUE 分支，一个 FALSE 分支，还有一个包含 8 种可执行行为的行为列表。参见图 12.15，图中的结构可能用来实现以下节点：

```
typedef struct DECNODE_TYP
{
    int operand1, operand2; // the first operands
    int comp1;             // the comparison operator
    int operator;          // the conjunctive operator
    int operand3, operand4 // the second pair of operands
    int comp2;             // the comparison to perform
    ACTION *act_true;      // action lists for true and false
    ACTION *act_false;
    DECNODE_PTR *dec_true; // branches to take if true or
                          // false
    DECNODE_PTR *dec_false;
} DECNODE, * DECNODE_PTR;
```

可以看出，这有很多小细节。如果这只有一个前提：

```
if (DAM < 100) then...
```

或变量和定量的差别：

```
if (DAM == FUEL) then ...
```

或

```
if (DAM == 20) then ...
```

然后确定（如果有两个前提或一个）：

```
if (DAM > 50) and (AMO < 30) then...
```

这些都是基础的程序问题，所以就不多介绍了。当你使用引擎读决策点、处理决策点时，要注意把上面这些考虑进去。总之，现在你了解了这部分内容，尝试写一个小决策树，使这个决策树能够在一些设定中决定做些什么。

我们现在来做一个开火控制树。你还没有真正做出一个完整的计划，但你可以认为下面这个例子是一个计划，因为例子中隐含的目标就是决定什么时候开火。这只有开火这一个目标。下面是我写的一个简要计划：

如果玩家距离近且自身当前损伤小则向玩家开火
 如果玩家距离远且剩余燃料多时寻找玩家
 如果当前损伤大且玩家距离近则躲避玩家
 如果当前损伤大且剩余弹药为 0 且玩家距离近则自毁

当然，一个完整的计划可能有几打或几百个这种语句。但是游戏设计者是用图形工具输入这些语句而不是把这些语句全都编写到程序中！这个计划的结果被转化为计划语言，最后的决策树如图 12.16 所示。

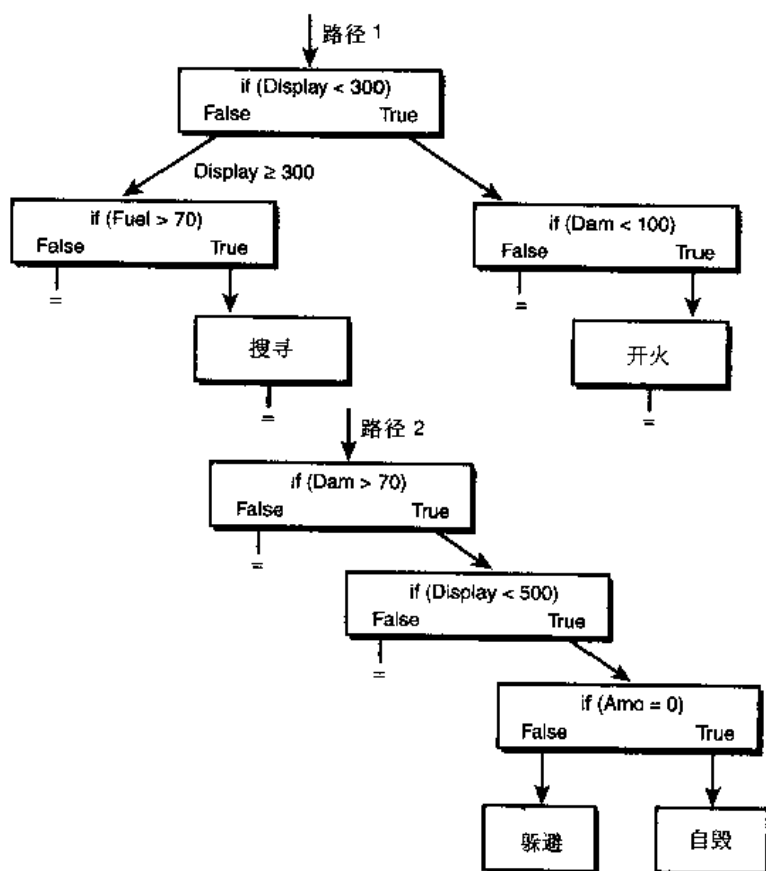


图 12.16 计划语言的最终决策树

是不是很简单？你只要编写一个处理器使它能够顺着树执行每个分枝即可。现在你知道了怎样创建一个能够处理决策和执行任务的决策树。在这小节结束之前你最好编写一个正式的计划算法，这个算法要把目标考虑进去，同时可以分析计划。

实现一个真正的计划者

你已经看到怎样可以实现计划者的条件部分，甚至运动部分。目标部分只是一种形式。每个具体的计划都有一个目标，并把这个目标设置为计划的终点。而且，当一个计划完成时，在计划的任何剩余部分能够被执行之前，必须测试目标。有时可能有与主计划同步执行的辅助计划，这些辅助计划必须有它们的目标，达到这些辅助目标可以促使主要目标实现。

例如，有一个总体计划——“所有的物体在基准点(x, y, z)集合”。除非每个物体都执行“到基准点(x, y, z)去”的计划，不然这个总体计划将会实现不了。而且，如果有个物体不能执行这个计划，那么计划者应该可以指出，并作出反应。这就是计划监督和分析的思想。我在这章的后面部分会具体介绍它。现在，我们来看看怎样表示一个计划。

计划本身可能隐含在决策/行为树中，或计划就是决策/行为列表，每个列表代表一个树或一个顺序。关键在于把计划、行为顺序、目标公式化。行为本身包含条件逻辑和低级的附属行为——如从(x,y,z)移到另一点或发射武器。在最高级的行为项是“杀死头目”或“占领堡垒”，而这可以直接通过引擎执行低等级的行为。

所以，假设一个计划是由一些数组或连接列表组成，同时你可以否认这个计划，计划者看起来如下所示：

```
while (plan not empty and haven't reached goal)
{
    get the next action in the plan
    execute the action
} // end while
```

当然，你应该知道这只是计划者的一个抽象工具。在真正的程序中，这些需要与其他的一些代码同步执行，而不可能等待一个单一计划达到目标后才运行：你必须用限态体、相似结构实现计划者并且在游戏运行时跟踪计划。

这个计划算法的缺点是算法非常迟钝。例如，将来的一些行为运行起来可能已经是行不通了，但这个算法并没有把这种情况考虑进去，因此计划就会失去作用。所以，计划者必须监控或分析计划。例如，如果计划要炸毁一座桥，而在路上其他人已经把桥炸毁了，计划者需要指出，并停止计划。这可以通过查看目标并测试任意目标是否由其他过程达到来实现。如果这个目标否认了计划或使计划无效，这个计划就应该停止。

计划者应该查看使计划实现不了的事件或状态。例如，计划中可能需要一张蓝图，但是蓝图已经丢失了。因此，找到蓝图这个目标就没有什么实际意义了。这类问题可以在现

行水平或将来水平中监控到，意思是计划者可以查看现在处在什么位置，或可以把自己置身于未来。例如，计划是“向东走 1000 英里，然后炸毁堡垒”。不要等物体走了 1000 英里后在准备炸毁时才发现炸弹已经用完了。计划者应该查看目标，并反向跟踪先决条件，测试物体是否有炸弹，或能否把炸弹带在路上。

另一方面，可能会发生意外。即使当时没有了炸弹，但可能在 1000 英里内有炸弹，所以只是由于一时缺乏资源就终止计划是不恰当的。我们可以按优先权来划分先决条件。例如，我想得到一枝激光枪，如果游戏中只有一枝激光枪而且被毁了，就没有必要继续执行这个计划。另一方面，如果我需要 100 个金币而现在只有 50 个，但我准备走一段路，在路上有很多方式可以得到金币，这就可以按计划执行。

最后，当一个计划出错了，你没有必要终止计划。你可以重新制定计划或选择另一个计划。对于每个目标你可以有 3 个计划，如果主要计划失败了，还有两个备用计划。

计划是功能十分强大的 AI 工具，可以运用到任何类型的游戏中。即使你只想写一个枪战游戏，你还需要一个可以影响游戏物体的“站在这个区域里，杀死玩家”目标的总体计划者。另一方面，像类似指挥和占领的战争模拟中，计划是使得游戏有意义的惟一方法！

把计划运用到真正的游戏中去的最好的方法是编写计划语言，然后给设计者一组变量和游戏物体，它们能够成为计划的一部分。这就使得设计者能够提出一些你从来没有想过的事情——硬件编码做不到这一点！

导航

导航就是从点 p_1 到目标 p_2 的路线的计算和执行。如图 12.17 所示。如果没有障碍物，简单的导向目标 AI 技术就足够了。然而，如果有障碍物，就必须有障碍回避。

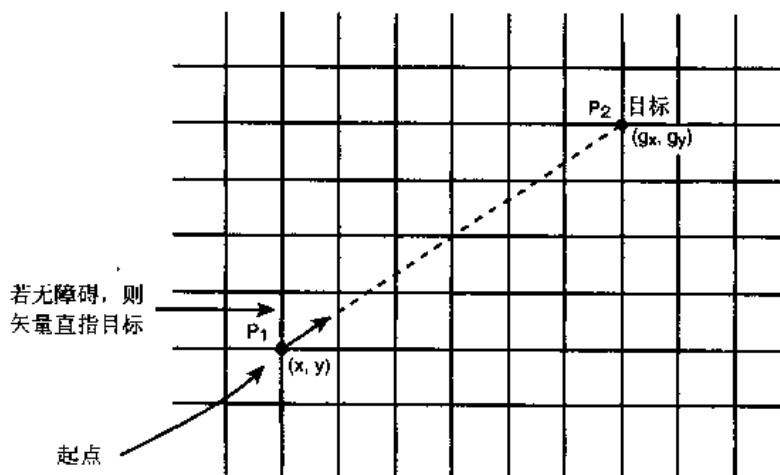


图 12.17 寻找点到点的路线

试错法

对于一些简单的障碍物（体积不是很大，大部分是凸多边形），通常用一个算法就可以避开障碍物——即当物体与障碍物相撞时，使物体后退，然后向左或右转动 $45^{\circ}\sim 90^{\circ}$ 度，然后继续向前运动一段固定的距离（`AVOIDANCE_DISTANCE`）。当物体走完这段距离，AI 把物体转向目标，使物体朝着目标继续运动。图 12.18 描绘了这个算法的运行结果。

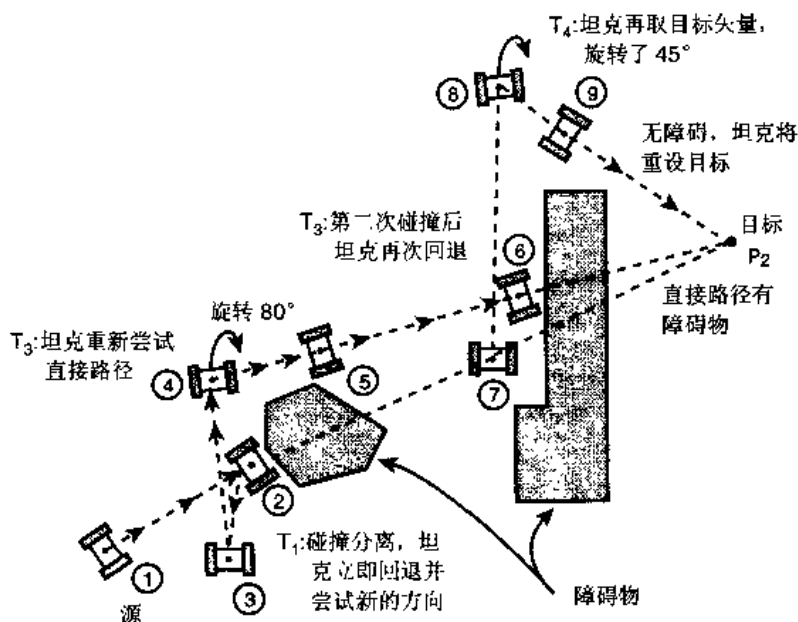


图 12.18 物体回避算法

这个算法的确不是很完善，算法能够实现避开障碍物是由于算法中包含了随机性。物体每次都是随机转动一个方向，所以物体迟早会找到绕开障碍物的路径。

轮廓跟踪

另一种避开障碍物的方法是轮廓跟踪。这个算法跟踪物体的轮廓——使物体顺着障碍物的轮廓线运行，定时测试物体所在位置到目标的直线是否与障碍物相交。如果没有交点，就可以直接向目标运动了，否则，继续跟踪。这个算法的结果如图 12.19 所示。

这个算法可以避开障碍物，但不是很灵活，因为算法只是沿着物体轮廓而不会取最近的路线。最好首先使用试错法，在一定时间后如果失败了，就转换到轮廓跟踪算法。

当然，通常玩家并没有像 3D 游戏 Quake 那样的“天眼”来观察视图，因此即使物体在避开障碍物时不是很灵巧，也不会很明显的显示出来。这只是用时稍长而已。但如果是具有俯视图的战争游戏的话，AI 操纵部队在追踪时看起来就的确很糟糕了。让我们看看能不能做得更好一些。

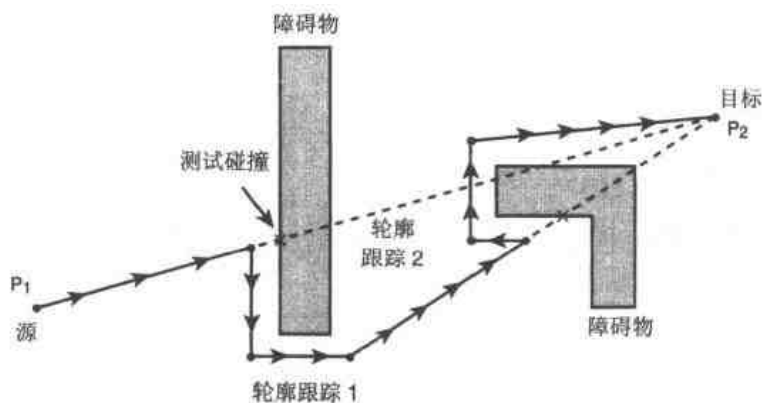


图 12.19 轮廓跟踪

避免碰撞轨迹

这个技术是在物体周围创建一个虚拟轨迹，轨迹由点或矢量组成。路线可以用最短路径算法（后面我们会介绍）计算出或用工具人工创建。

在每个大障碍物周围是一条可见轨迹，但这条轨迹只有导航 AI 看得见。当游戏物体想避开障碍物时，物体就会请求对于这个障碍物而言最短的避开路线，然后得到这个路线。这就保证了物体总能知道怎样避开障碍物。当然，你可能想让每个障碍物有多个避开路线，或加上些跟踪“干扰”使得物体不是完全按避开路线运行。如图 12.20 所示。

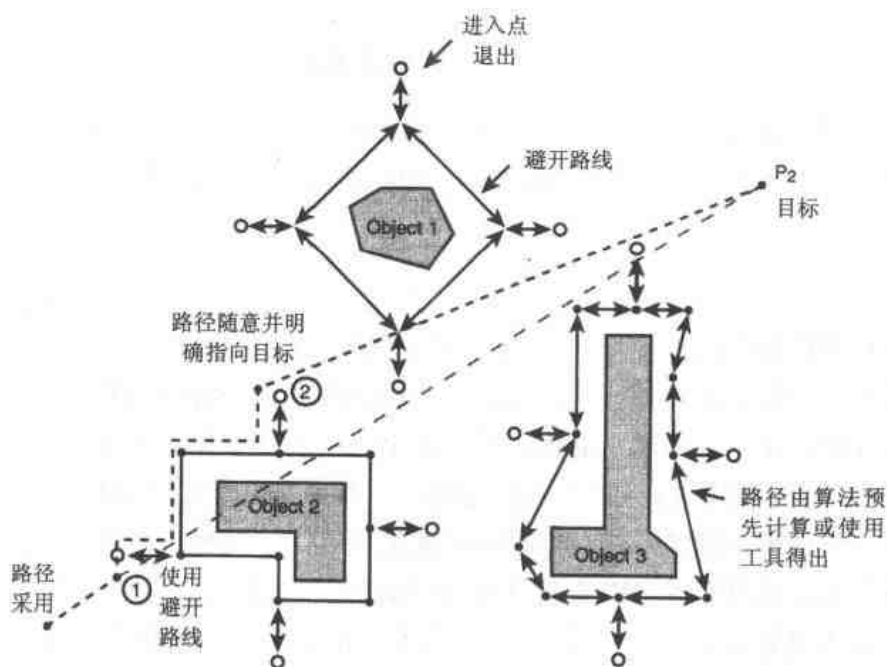


图 12.20 物体避开路线

这会让我们想到另一个问题：为什么在游戏中不能有几十或几百条预先计算过的路线呢？那么，当物体从点 p_i 到点 p_j ，不用导航和避开障碍物，只用沿着一条预先计算的路线运行就可以了。

导航基准点

假设有一个复杂的世界，里面有很多各种各样的障碍物。当然，你能够创建一个智能物体，这个物体能够自己导航，可以绕过各种障碍物，最后到达目的地。这是必要的吗？不是！你可以建立一个能够连接游戏中所有重要点的路线网，路线经过一个互相连接的节点网。每个节点代表一个基准点或游戏中一个重要的点。网络中的线代表矢量方向和网络中一个点到另一个点的长度。

例如，假设你想使一个物体从桥上运动到城里。这是不是看起来很复杂？但是如果你有路线网，并找到经过桥到城里的路线。你只要使物体顺着这条路线运动就可以了。物体肯定能够避开所有的障碍物并到达目的地。图 12.21 就是上面所说的具有路线网的复杂地区的俯视图。箭头线就是我们所要找的路线。记住，这个路线网不仅仅使物体避开障碍物，而且还有到重要目的地的路线。

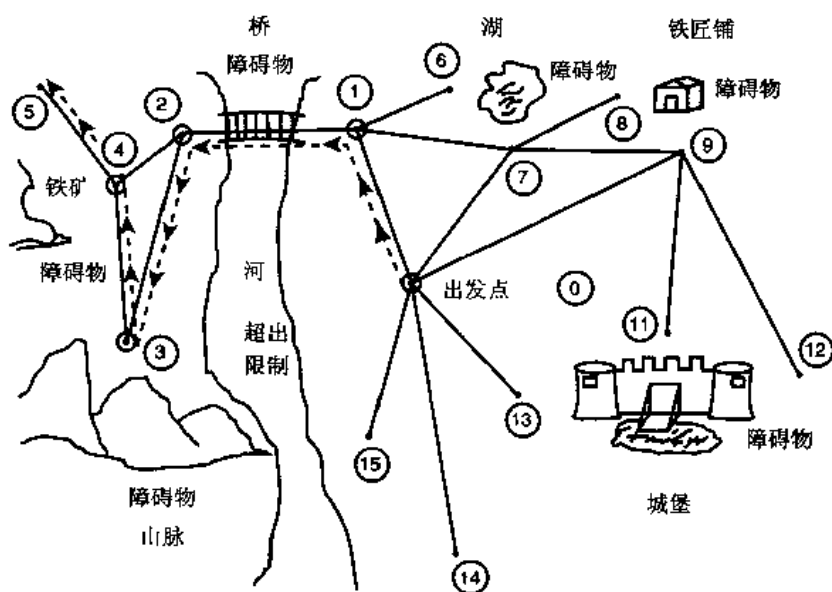


图 12.21 导航网

这两个问题：1) 沿着路线移动；2) 表示路线网的数据结构可能有些难以处理。首先，我们来谈谈第一个问题。

假设有一条从点 p_1 到点 p_2 的路线，由 n 个节点构成，可以用下面的数据结构表示：

```
typedef struct WAYPOINT_TYP
{
```

```

int id;           // id of waypoint
char *name ;     // name of waypoint
int x,y;         // the position of waypoint
int distance;    // distance to next waypoint on path
WAYPOINT_PTR *next ; // next waypoint in list
} WAYPOINT

```

这只是一个例子；你可以使里面的内容完全不同。现在假设有 5 个基准点，包括起点和终点 p1 和 p2，如图 12.21 所示。这些基准点表示为：

```

WAYPOINT path[5] = { { 0,"START",x0,y0,d0,&path[1]},
                      { 1,"ONPATH",x1,y1,d1,&path[2]},
                      { 2,"ONPATH",x2,y2,d2,&path[3]},
                      { 3,"ONPATH",x3,y3,d3,&path[4]},
                      { 4,"ONPATH",x4,y4,d4,NULL}};

```

首先应该注意到虽然我将基准点放置在数组里，但仍然可以把它们的指针连接起来。同样，最后的连接是 NULL，因为这是终点。

沿着路线移动时，还有几件事需要考虑。首先，必须要到达路线的第一个节点上，或沿着路线的某一节点。这可能是个问题。假设在游戏网格中有足够的路线进入这个点，你可以假设一个路线节点在路线进入点的范围内。找一个最近的节点，然后朝这个节点移动。在最初的路径定位时，你必须使路线避开障碍物！一旦游戏物体处于起点或路线中的一个节点上时，就可以准备开始沿着路线移动了。

沿着路线运动

路线是由一连串的点组成，这一串点总是一个点到另一个点，而中间肯定没有障碍物。所以，你可以很容易就把物体从一个基准点移到下一个点上，然后再往下一个点移动，直到最后一个基准点（目的地）：

```

find nearest WAYPOINT in desired path
while (not at goal)
{
    compute trajectory from current waypoint to next
    and follow it.
    If reached next waypoint the update current
    waypoint and next waypoint.
} // end while

```

也就是你只要使物体沿着这些点移动直到最后一个点即可。如果想得到从一个基准点到下一个点的轨迹矢量，可以使用下面的代码：

```

// start off at beginning of path
WAYPOINT_PTR current =&path[0];

```

```
//find trajectory to next waypoint
trajectory_x  path->next.x - path->x;
trajectory_y  path->next.y - path->y;
// normalize
normalize(&trajectory_x,&trajectory_y);
```

想要使轨迹（两点之间）标准化，必须使轨迹长度等于 1.0——用矢量长度划分每部分（参见附录 C “数学和三角函数回顾”）。只要得到物体的轨迹方向，等待物体移到下一个基准点，然后继续这个算法。当然，我省去了一些细节。（例如当到达一个基准点时会发生什么事情？）我建议检查物体与基准点的距离。如果物体与基准点的距离足够近，则可以选择下一个基准点。

关于路线，这里有几个问题。首先，找到一条合适的路线可能和从一个点移到另一个点一样困难！这是个问题，但是用适合的网络数据结构，你就可以保证对于任意给定的游戏单元格，一个游戏物体只需要移动大约 100 个单元就可以选择一条路线。由于有些连接可能被几条路线共享，所以用数据结构代表路线网会比较复杂。但这主要还是数据结构方面的问题，取决于你怎样设计。你可以只是设置 1000 个不同的路线，不重复使用基准点，即使一些路线用到同样的基准点也无妨。

或者你可以用一个能够重复使用基准点的连接图表，这个连接图表用逻辑和数据连接来构成一条路线，而不会转变轨迹。这可以通过灵活的指针运算或逻辑选择构成的特定路线的正确连接得以实现。

例如，如图 12.22 所示，图中有两条经过相同基准点的路线。你需要将物体可能到达并在相关线路上的基准点编译成一系列——如果想使物体移到一个目的地，物体所在的路线基准点有 16 个引出连接，选择一个目的地在终点的连接线路。选择路线取决于你自己，怎样实现则取决于游戏中的环境。

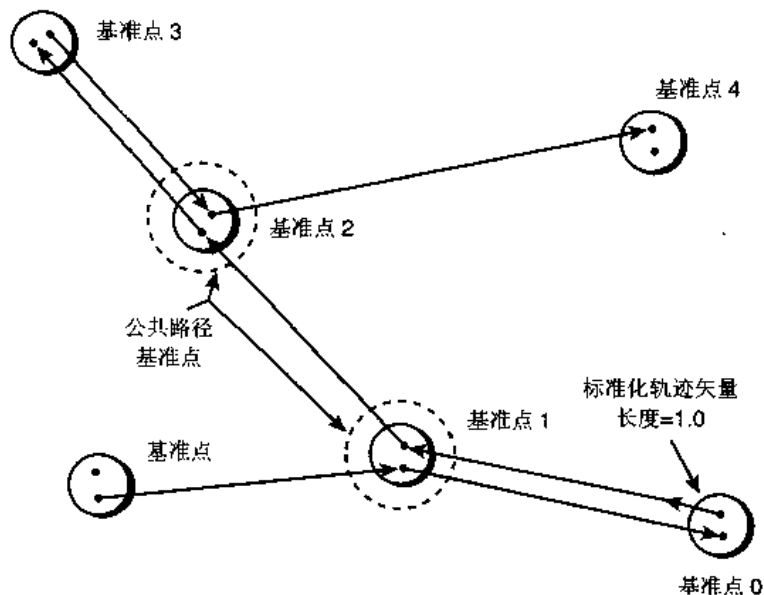


图 12.22 有公共基准点的路线网

飞车实例

使用路线最好的例子是飞车游戏。假设有很多汽车在一个跑道上，这些汽车绕着跑道行驶并且能够避开玩家，而且这些汽车比较智能化。

你需要创建 8 条或 16 条沿着跑道的路线。每条路线等距或属性稍微有些不同。每个 AI 汽车从不同的路线出发，出发后沿着各自的路线行驶。如果有一辆汽车发生了碰撞，它会及时选择一条最近的路线，接着行驶。

另外，如果汽车 AI 可以把路线转换为更为进取的路线。这会有很大的帮助，因为你不用当心汽车会聚在一块，也不用过多的控制这些汽车。控制汽车的速度和刹车时间会使得游戏更具真实感。

参见 CD 中的 DEMO12_8.CPP.EXE。上面用单一基点路线创建了一个小型飞车演示。汽车避免撞到其他汽车，但两辆汽车即使相互接触，也不会撞毁。这是个 DirectX 应用程序，所以你必须加上库，并编译。

稳定的导航

我想简单谈谈真正的导航——利用计算机科学算法找到从点 p_1 到点 p_2 的路线。有几种算法可以实现这个功能。问题是没有一个算法是实时的，因此不适用于游戏。但是，如果使用一些技巧，就可以使这些算法变成实时的，并能够运用到游戏中。当然，你可以在工具中用这些算法计算路线。

所有这些算法都作用于图形状结构，这些结构代表由节点和边组成的游戏环境，边是由点组成。每条边都有一个值。图 12.23 显示了典型的图形。

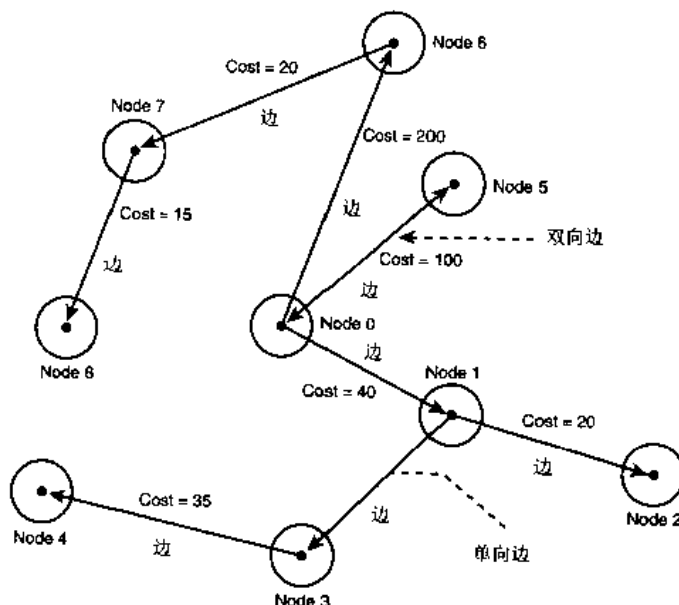


图 12.23 典型的图形网

由于讨论的是 2D 和 3D 游戏，所以你可以认为图形只是一个网格，这个网格处于游戏区域里，游戏区域中每个单元格隐含地与周围的 8 个单元格相连，值代表距离的长度。如图 12.24 所示。

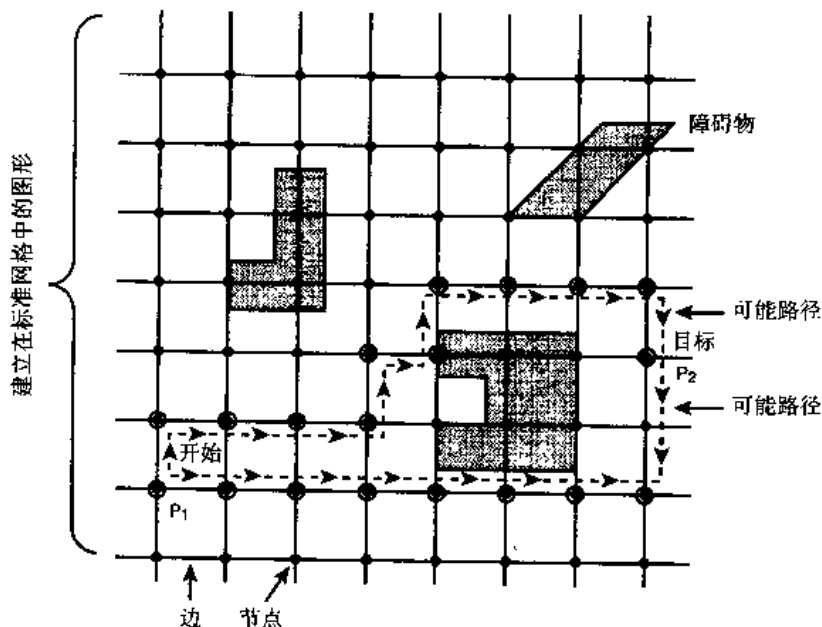


图 12.24 用游戏环境中的标准网格建立一个图形

总之，已经知道怎样表示一个图形状的游戏世界。可以用这个算法找到从点 p_1 到点 p_2 的一条较短的路线，或最短路线（避开障碍物）。图形中不允许出现障碍物，所以障碍物不会是路线的一部分。

有几种导航算法非常有名，下面列出了这些算法，并简要介绍这些算法：

- 宽度优先搜索
- 双向宽度优先搜索
- 深度优先搜索
- Dijkstra 搜索
- A* 搜索

宽度优先搜索

这种方式同时搜索所有方向，遍历一个单位距离远的节点，而后两单位距离远的节点，三单位……是一个逐渐增大的圆。这个方法不灵活，因为宽度优先搜索没有把遍历集中在目标实际方向上。如图 12.25 所示，宽度优先搜索的算法：

```
void BreadthFirstSearch(NODE r)
{
    NODE s; // used to scan
```

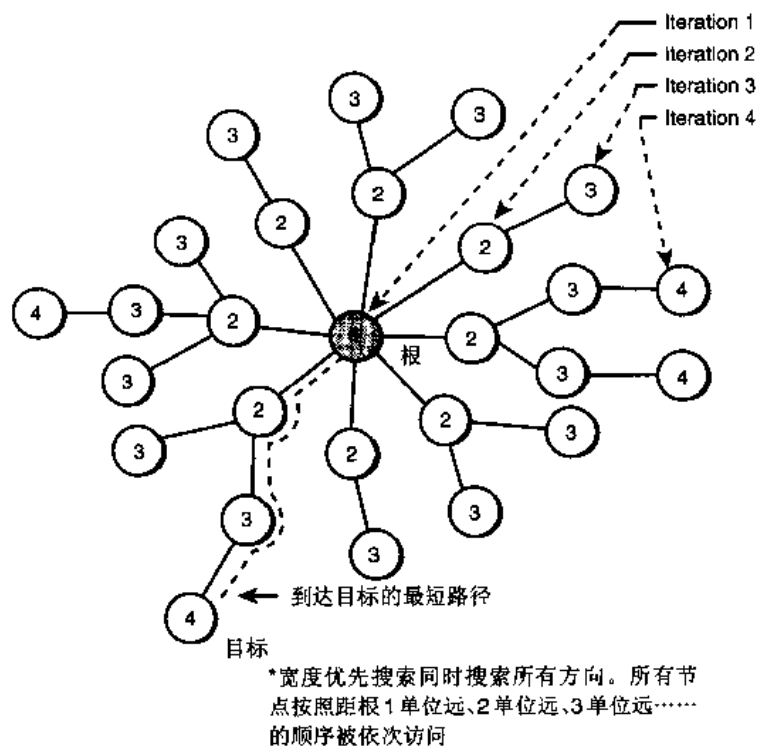


图 12.25 宽度优先搜索

```

QUEUE q; // this is a first in first out structure FIFO

// empty the queue
EmptyQ(q);

// visit the node
visit(r);
Mark(r);

// insert the node into the queue
InsertQ(r,q);

// while queue isn't empty loop
while (q is not empty)
{
    NODE n = RemoveQ(q);

    for (each unmarked NODE s adjacent to n)
    {
        // visit the node
        Visit(s);
        Mark(s);

        // insert the node s into the queue
    }
}

```



```

    InertQ(s,q);
  }// end for

  }// end while
} // end BreadthFirstSearch

```

双向宽度优先搜索

双向宽度优先搜索与宽度优先搜索相似，但双向宽度优先搜索包含两个不同的宽度优先搜索：一个在出发点，一个在终点。当搜索重叠时，就会计算出最短的路线。如图 12.26 所示。

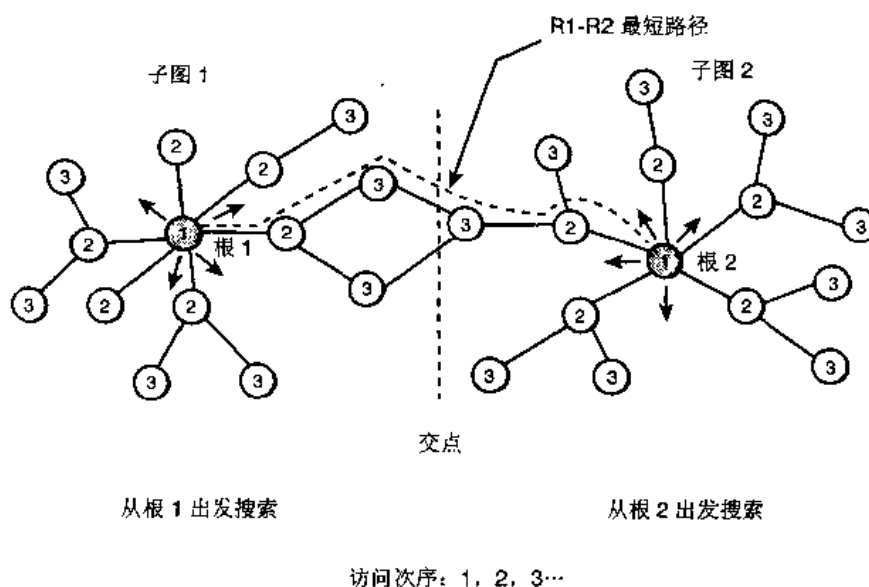


图 12.26 双向宽度优先搜索

深度优先搜索

深度优先搜索与宽度优先搜索相反。深度优先搜索自始至终搜索一个方向，直到搜索完这条路线或找到目标为止，然后搜索下一个方向，依次类推。深度优先搜索的问题是需要一个停止界限。也就是“物体与目标距离为 100 单元，而正在搜索的路线的值已经为 1000，这就需要转到另一条路线上”。参见图 12.27，深度优先搜索的算法如下：

```

void DepthFirstSearch(NODE r)
{
    NODE s;    // used to scan

    // visit and mark the root node
    visit(r) ;
    mark@;
}

```

```
// now scan along from root all the nodes adjacent
while(there is an unvisited vertex a adjacent to r)
{
    DepthFirstSearch(s);
} // end while
} // end DepthFirstSearch
```

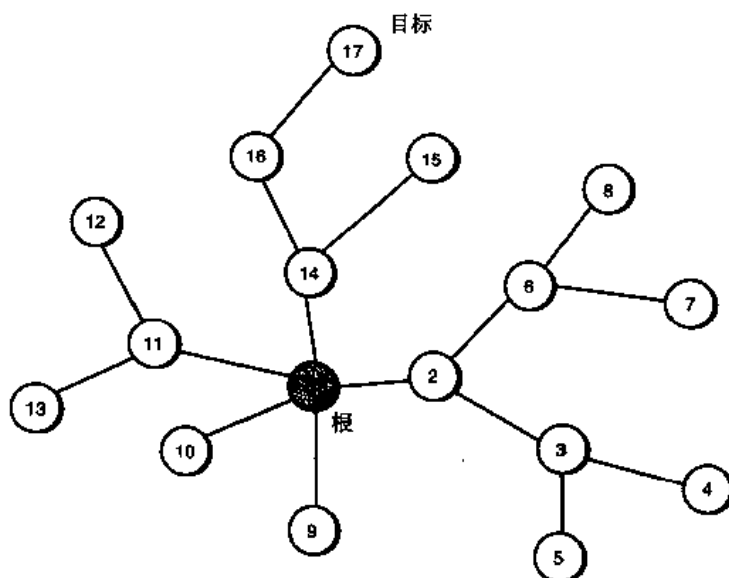


图 12.27 深度优先搜索

Dijkstra 搜索

Dijkstra 搜索源于查找最小生成树的算法。在每次迭代中，算法确定到下一点的最短路线，然后沿着这条路线移动，而不是盲目地沿着一个随机方向运行。

A* 搜索

A* 搜索与 Dijkstra 搜索相似，不过 A* 搜索采用直观推断法，这不仅查看从起点到所在位置点的距离值，而且估计从所在位置点到目的地的距离，即使不遍历路线上的节点也可进行推断。

当然，可以改进这些算法或混合使用这些算法。有一些相关的书籍介绍了这些算法的实际运用，如果你有兴趣，可以翻阅一下。

技巧



最初的算法也可以查找最好的路线。这些算法可能不是实时的，但这些算法的定义比其他算法更加容易理解。

高级 AI 脚本

现在，你应该对 AI 的一些基础知识有了认识，接下来我要用这些基础知识来谈谈高级的 AI 脚本。

在这章前部分，学习使用简单的[OPCODE OPERAND]语言和虚拟解释程序时，已经知道脚本命令语言可以运用到 AI 中。当然，这是脚本的一种形式。我要介绍另一种创建决策树的方法——用基于逻辑、输入、运算符、行为函数的脚本语言创建决策树。QUAKE C 和 UNREAL Script 是使用这个技术的很好例子。它们都允许你使用类似英语的高级语言编写游戏代码，这些高级语言由引擎执行。

设计脚本语言

脚本语言的设计是基于你想实现的功能。这有几个问题需要问自己：

- 脚本语言只用于 AI 还是准备用于整个游戏？
- 脚本语言要编译或解释吗？
- 脚本语言是超高级语言，采用和英语类似的语法，还是低级的采用函数、变量、条件逻辑等的类编程语言？
- 是不是所有的游戏都用脚本语言设计？也就是，程序员做一些的硬件代码设计，还是这个游戏完全用脚本运行？
- 你想使得游戏设计者或脚本设计者有多复杂，功能有多强大？设计者是否可以利用系统或引擎变量？
- 使用脚本语言的程序员的水平有多高？是 HTML 编码者，还是编程初学者，或专业的软件工程师？

在你开始设计语言时应该思考一下这些问题。当你能够回答这些问题时，你就可以使用脚本语言真正开始设计整个游戏了。

这是很重要得一个阶段。如果用脚本语言完全控制游戏，所用的脚本语言最好可扩充、稳定、可延伸，而且功能强大。例如，脚本语言应该能够建立一个时常飞行的飞机，以及一个向你进攻的怪物。

总之，记住脚本语言的思想就是在引擎上创建一个高水平的界面，这样就不用编写低级 C/C++ 代码来控制游戏物体了。解释性的或编译过的伪英语语言能够描述游戏中的行为。如图 12.28 所示。

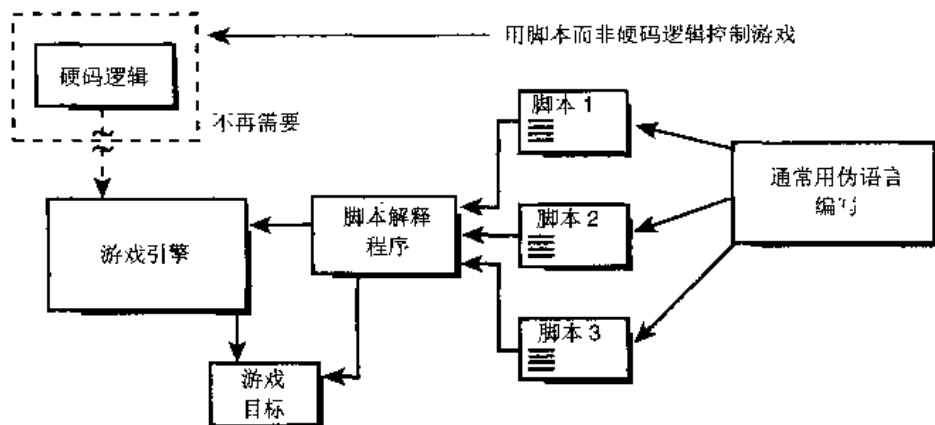


图 12.28 引擎和脚本语言的关系

下面是控制街灯的虚构脚本语言：

```

OBJECT: "Street Light",SL
VARIABLES:
    TIMER green_timer; // used to track time
    // called when object of this type is created
ONCREATE()
BEGIN
    //set animation to green
    SL.ANIM = "GREEN"
END
// this is the main script
BEGINMAIN
    // is anything near the streetlight
    IF EVENT("near","Street_Light") AND
        TIMER_STATE(green_timer) EQUAL_OFF THEN
        BEGIN
            SL.ANIM = "RED"
            START_TIMER(green_timer,10)
        ENDIF
        // has the timer expired yet
        IF EVENT(green_timer,10) THEN
            BEGIN
                SL.ANIM = "GREEN"
            ENDIF
        ENDIF
    ENDMAIN

```

虽然上面的代码可能有些漏洞，但关键是这代码水平非常高。关于设置动画，检查接近度等细节问题有很多个，但使用这个语言，任何人都可以编写出交通灯程序。

代码启动，用 ONCREATE（）把灯设置成绿色，然后用 EVENT（）测试是否有东西

接近，如果是，把灯的颜色转为红色。红灯停留一段时间后，又转为绿色。这个语言看起来有点像 C、Basic、Pascal 的混合！

这就是你需要用来设计和实现控制一个游戏（不确定的物体知道怎样操作任何游戏物体）的语言。例如，当使用 BLOWUP（）时，语言处理程序最好知道怎样使游戏中任意物体运行。即使调用消灭蓝色妖怪的函数可能是 TermBMs3()，调用炸毁围墙的函数可能是 PolyFractWallOneSide()，你也只须调用 BLOWUP("BLUE")及 BLOWUP("wall")就可以了。弄明白了吗？

你可能不知道怎样使用这些脚本语言中的一个。这的确不是很容易。首先，你必须确定要使用的是解释过的语言（用游戏引擎中的解释程序解释的语言）或编译语言（将要编译成标准代码的语言）还是想使用处于两种语言之间的语言？然后你必须编写语言、语法分析程序、代码生成程序、P 代码解释程序，或使代码生成程序生成 PentiumX 机器码或转换成 C/C++。

LEX 和 YACC 可以帮助你实现这些功能，这两个工具支持词法分析程序和另一个编译程序的编译程序。它们是语言分析和定义工具，能够帮助你实现递归分析程序及复杂的编译程序或解释程序所需要的状态机。我写的《Sams Teach Yourself Game Programming in 21 Days》这本书介绍了一些语言分析。你也可以到网上查看相关的资料。

使用 C/C++编译程序

使用解释语言的好处是引擎能够读它，而且游戏不需要重新编译。如果你不介意编译，就可以使用初等游戏脚本语言：用 C/C++预处理程序来转换脚本语言。预处理程序只转换与编译程序无关的头文件和 C/C++源程序。

C/C++预处理程序真的是令人惊异的工具。它能够进行符号引用、置换、比较、计算、等。如果你不介意用 C/C++作为最底层的语言并编译脚本，你可以通过巧妙的设计（比如置换许多文件、用许多已经存在的函数、识辨物体、好的物体导向设计）来编写脚本语言。

说清楚这个问题的最好的方法用一个简单的例子，首先，脚本语言必须编译，只要创建需要引用的物体，每个脚本就开始运行。当物体消失后，就会停止运行脚本。我们所要编写的脚本语言基于 C，所以就不用复习了。

脚本由下面几部分组成：

全局部分——这里定义脚本中所有的全局变量。只有两种数据类型：REAL 和 INTEGER。REAL 与 C 语言中 float 类似，INTEGER 与 C 语言中的 int 类似。

函数部分——这部分由函数组成。里面的所有函数的语法为：

```
data_type FUNCNAME(data_type parm1, data_type parm2,...)
BEGIN
    // code
ENDFUNC
```

主体部分——程序从这部分开始执行，然后循环调用这部分，直到游戏物体消失：

```
BEGINMAIN
// code
ENDMAIN
```

至于变量的赋值和比较，只有下面的语法才是合法的：

赋值 变量=表达式

等于 (表达式 EQUALS 表达式)

不等于 (表达式 NOTEQUAL 表达式)

比较——大于，小于，大于等于，小于等于，与 C 标准一样，如下：

(表达式 > 表达式)

(表达式 < 表达式)

(表达式 >=表达式)

(表达式 <=表达式)

条件——条件语句的形式和 C 一样，只是条件语句后面的执行语句必须在 **BEGIN** **ENDIF** 块中，如下：

```
if ( a EQUALS b)
BEGIN
// code
ENDIF
else
BEGIN
// code
ENDELSE
```

类似的，**else** 和 **elseif** 块必须被包含在 **BEGIN** 和 **ENDELSE** 块中。

这里没有转换语句，只有一种循环语句，为 **WHILE** 循环：

```
WHILE(condition)
BEGIN
// code
ENDWHILE
```

还有一个 **GOTO** 关键字可以从一条语句跳转至另一条语句。跳转必须标明所要跳转到的语句的名称：

```
LBL_NAME:
```

NAME 是长度小于 16 位的字符串。如下：

```
LBL_DEAD:
if ( a EQUALS b) BEGIN
```

```
GOTO LBL_DEAD;
ENDIF
```

现在，你可能已经抓住要点了。当然，你可以添加几十个甚至几百个高级辅助函数用来检查游戏物体。比如，如果游戏物体有健康状态或生命状态，可以添加一个 **HEALTH()**：

```
if (HEALTH(* alien *) >50)
    BEGIN
    // code
    ENDIF
```

而且，可以创建一个能用正文串参数测试的事件：

```
if( EVENT( * player dead *)
BEGIN
// code here
EDNIF
```

通过使用灵活的全局状态变量，确保脚本接受足够的普通事件，同时系统状态变量（通过函数）和许多文本替换能通过预处理程序实现这些所有功能。省去一些细节使问题简单一些，想一想你需要文本替换做些什么。参见图 12.29，每个编译过的脚本首先用 C/C++ 预处理程序进行处理。

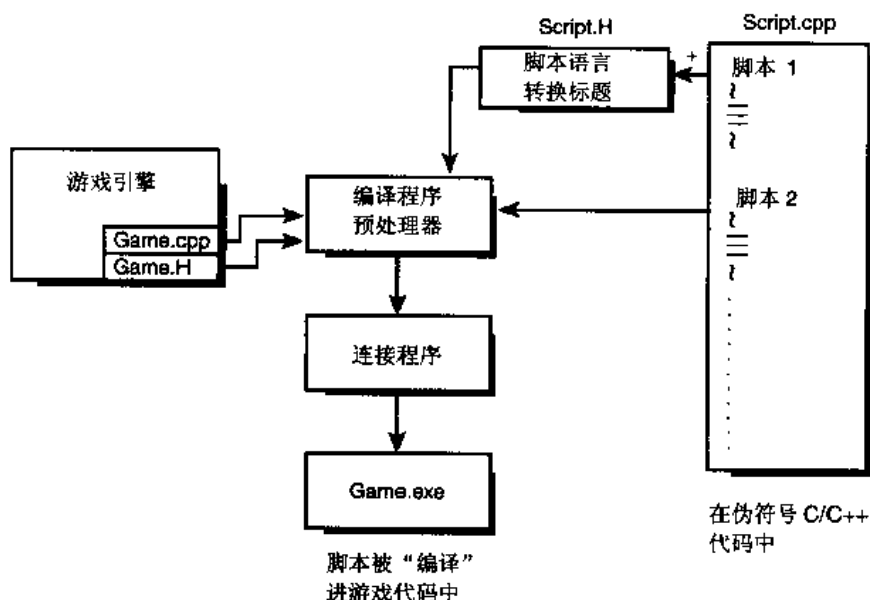


图 12.29 用 C/C++ 预处理程序解释脚本语言

由此可以得到所有的文件替换，并将很少一部分脚本语言转换回 C/C++。想要使脚本

可以运行，必须保存所有的文件和扩张脚本 .SCR，当文件输入到主 C/C++ 文件中进行编译时，必须保证文件中包含了脚本转换标题。下面是脚本翻译程序：

```
SCRIPTTRANS.H
// variable translation
#define REAL static float
#define INTEGER static int
// comparisons
#define EQUALS ==
#define NOTEQUAL !=
// block starts and ends
#define BEGIN {
#define END }
#define BEGINMAIN {
#define ENDIF }
#define ENDWHILE }
#define ENDELSE }
#define ENDMAIN }
// looping
#define GOTO goto
```

然后，把下面这条语句添加到游戏程序中：

```
#include "SCRIPTTRANS.H"
```

然后，在游戏引擎中的某处添加上实际脚本文件。可以添加在开头部分，也可以添加在函数中：

```
Main_Game_Loop()
{
#include "script1.scr"
// more code
} // end main game loop
```

这部分可按你的意愿完成。关键是编写的代码必须以某种形式编译，代码必须能够贯穿全局、了解事件、调用函数。下面是一个初级脚本，这个脚本可以触发一个事件：

```
// the variables
INTEGER index;
index = 0;
// the main section
BEGINMAIN
LBL_START;
if (index EQUALS 10)
BEGIN
BLOWUP("self")
ENDIF
```



```

if(index < 10)
    BEGIN
        index = index + 1
        GOTO LBL_START
    ENDIF
ENDMAIN

```

显然，必须要定义 BLOWUP()。预处理程序将这些代码转化为：

```

{
    static int index;
    index = 0;
    LBL_START:
    if(index == 10)
    {
        BLOWUP("self");
    }
    if (index < 10)
    {
        index = index +1;
        goto LBL_START;
    }
}

```

当然，我省去了许多细节，比如变量名抵触问题、访问全局变量、调试、确保脚本不是处于死循环状态等等。

提示

Visual C++ 编译程序可以用编译指示语句输出预处理过的 C/C++ 文件。

人工神经网络

你可能经常听说神经网络，在最近 3~5 年里，人工神经网络的研究有了很大的发展。不是因为有了什么突破性的成就，而是人们对人工神经网络越来越感兴趣，因而越来越多地研究并使用人工神经网络。实际上，一些游戏成功地使用了最先进的人工神经网络技术，如：Creatures、Dogz、Fin Fin 等。

神经网络是人脑的模型。人脑是由 100 亿~1000 亿个脑细胞组成。每个脑细胞都能够处理和发送信息。图 12.30 为一个脑细胞或神经元的模型。

神经元的 3 个主要组成部分为：体质、轴突、树突。体质是主要的胞体，用于处理信息。轴突传送信息给树突，而树突把信息传递给其他神经元。

每个神经元都有相当简单的功能：处理输入信息，发射或不发射。发射的意思是发送

一个电化学信号。神经元有一些输入端，只有一个输出端，并有处理输入和产生输出的规则。规则十分复杂，简单的说就是神经元不断累积出现的信号，累积到一定程度会促使神经元发射。

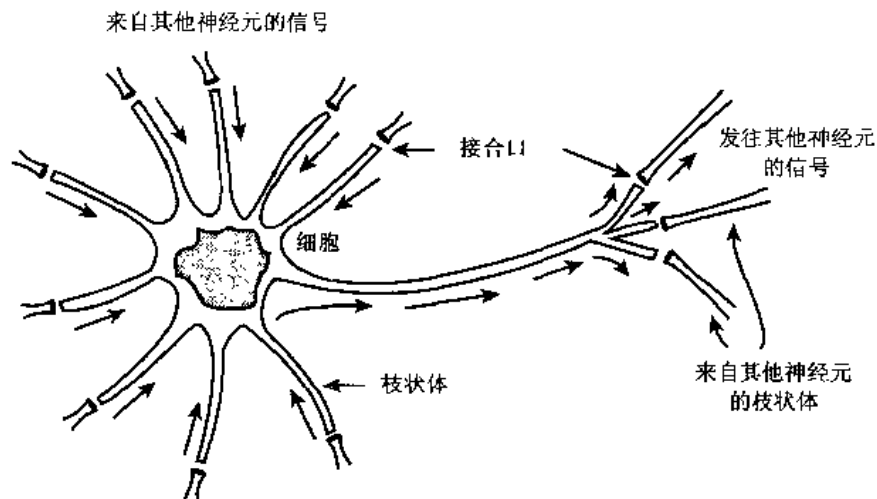


图 12.30 一个基本神经元

但是，这些怎样运用到游戏中呢？我们并不是要试图创建一个和思想或意识一样难以琢磨的东西，我们可以从简单的记忆、模式识别开始，学习一些计算机模型。我们的大脑非常擅长这些工作，但计算机并不擅长。开发与人脑相似的生物计算机是很具吸引力的。

人工神经网络（或简单的神经网络）是可以并行处理信息的简单模型，就像我们的大脑一样。我们来看看人工神经元最基本的性质。

第一个人工神经网络是 McCulloch 和 Pitts 在 1943 年创建的。McCulloch 和 Pitts 都是电气工程师，当时他们想仿效人脑制作一个电子部件。他们称之为神经元，如图 12.31（左）所示。现在的神经元变化很大，如图 12.31（右）所示。

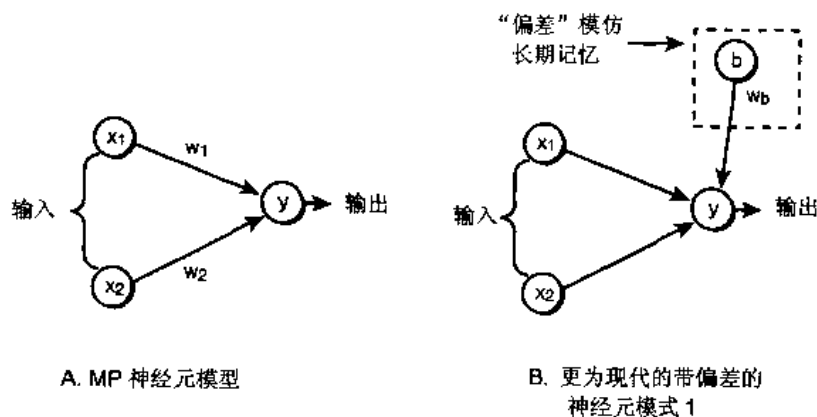


图 12.31 人工神经元

一个神经元由几个输入组成，这些输入 $X(i)$ 与总权重 $W(i)$ 成比例，然后用一个激活函数进行处理。这个激活函数可能是一个简单的临界值，如在 McCulloch-Pitts (MP) 模型中，也可以是一个比较复杂的阶跃函数、线性函数或指数函数。在 MP 模型中，总权重值和临界值相比较，如果总权重值大于临界值，神经元就发射；否则，就不发射。用数学公式表示为：

McCulloch-Pitts 求和函数

$$\text{Output } Y = \sum_{i=1}^n X_i \cdot W_i$$

通常神经元有偏差

$$\text{Output } Y = B \cdot b + \sum_{i=1}^n X_i \cdot W_i$$

现在来看看一个基本的神经元是怎样工作的。假设神经元有两个输入 X_1 和 X_2 ，都采用二进制值 0, 1。然后把临界值设置为 2, $w_1=1$, $w_2=1$ 。求和函数：

$$Y = X_1 \cdot w_1 + X_2 \cdot w_2$$

比较 Y 和临界值。如果 Y 大于或等于 2，神经元就会发射，输出一个 1.0；否则，输出一个 0。表 12.3 是一个描述单一神经元运行的真值表。

表 12.3 单一神经网络的真值表

X_1	X_2	总和	最后输出
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

如果仔细观察这个真值表，你会发现这是个与门电路。所以一个小神经元能够执行“与 (AND)”操作。实际上，用神经元你可以建立任何想要的逻辑电路。如图 12.32 所示。图中显示了一个 AND 电路，一个 OR 电路，一个 XOR 电路。

当然，真正的神经网络要复杂得多。它们有很多层，复杂得多的激活函数，几百个甚至几千个神经元。但现在，至少你已经了解了神经网络的基本构件。神经网络正在带给游戏前所未有的冲击。不久游戏就会有决策和学习功能！

这是个有趣的领域，但限于篇幅，我只能介绍到这。不过，CD 中在 ARTICLES\AINETWARE\目录下有一篇关于神经网络的文章，它会使你了解更多的相关知识。文章中覆盖了所有的网络种类，介绍了学习算法，举例说明了神经网络能够做些什么。文章中还有所有

的源代码，这些代码是可执行代码，并有两个版本——MS-Word.DOC 版本和 Adobe Acrobat.PDF 版本。

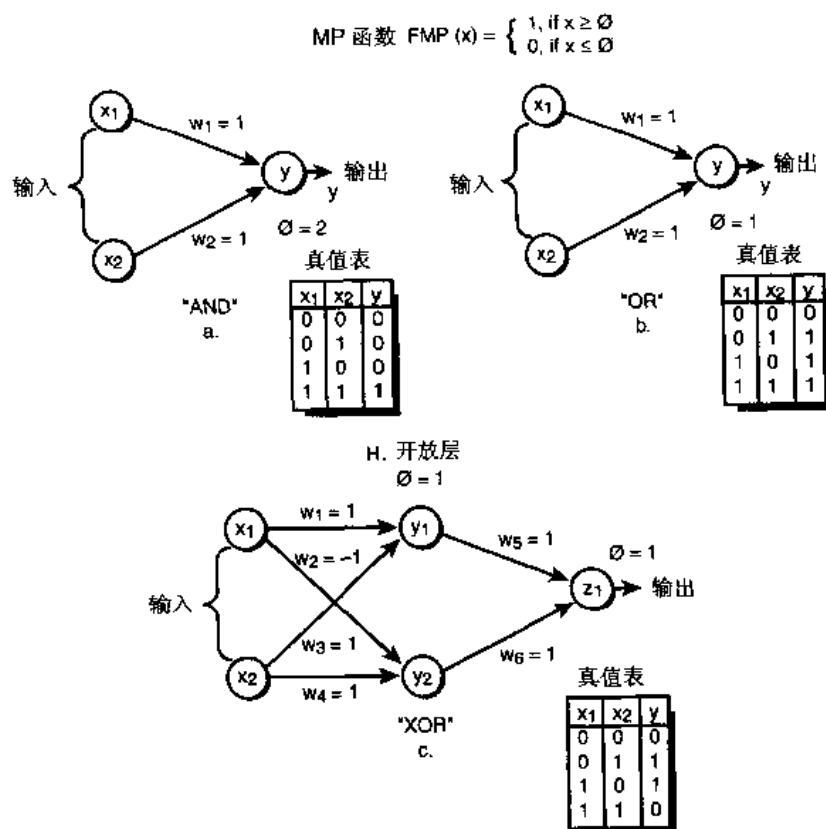


图 12.32 基本逻辑电路

遗传算法

遗传算法是依赖生物模型演算的一种计算方法。自然在演化中很重要，遗传算法试图捕获一些自然进化的本质，遗传进化用于计算中，能够帮助解决一些用标准方法解决不了的问题。

遗传算法工作方式为：将一些信息指示符作为字符串放入到一个位向量中，像一股 DNA 一样。如图 12.33 所示。

这个位向量代表策略，或一个算法的编码，或解决方案。我们需要从一些位向量开始工作。然后处理位串和一切它通过记录其合适度的目标函数所代表的东西。结果就是记录。位向量是不同的控制变量或一些算法的调整的连接，然后根据直觉或先验知识，人工提出一些实验设置值。运行每个设置值，得到相应的记录。

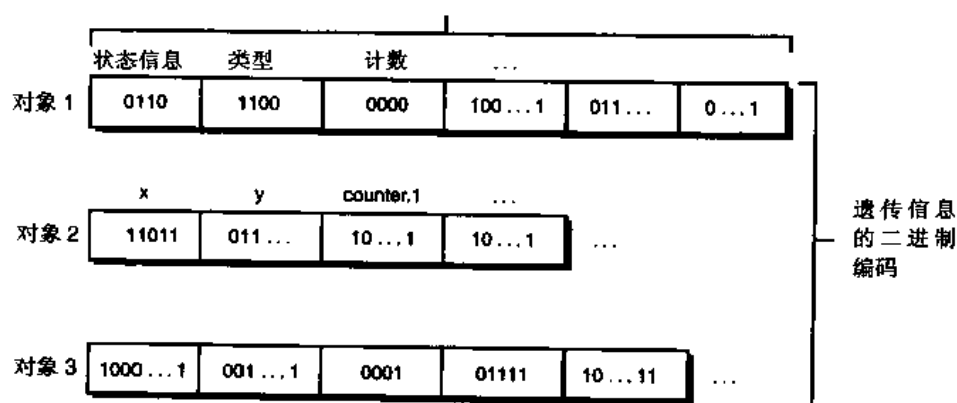


图 12.33 遗传信息的二进制编码

只要所用的方法是正确的，或能够正确使用遗传算法，就可以转换解决方法。混合使用两种解决方法或控制所有向量来创建两个新的向量。如图 12.34 所示。

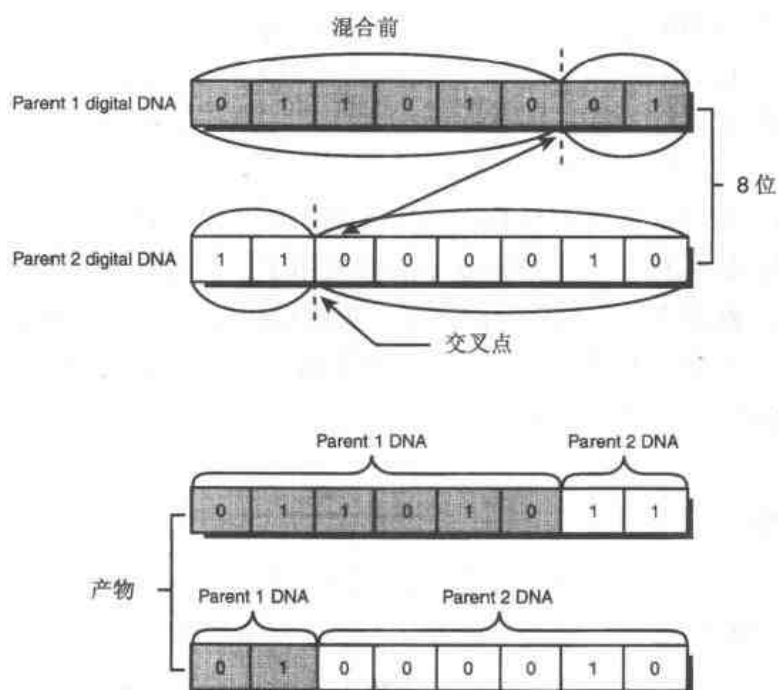


图 12.34 数字结合

在转换过程中模拟变化时，添加一些不确定性。在使用新的解决方法时，同时采用最近产生的最好的方法，看看记录有什么变化并挑选出最好的结果，重新执行上面的步骤。最好的解决方法不容易得到，非常费时，但结果可能是出乎意料地好。

遗传算法中最关键的是算法中采用新的模式和要查找非常大的空间（一般来说如果是一步一步查找是找不到的）。这是由于突变的缘故，突变代表完全随机事件，这些事件有可能产生更合适的结果。

怎样运用到游戏中呢？有很多种方法。但我只引你入门。在数字 DNA 中，你可以把 AI 的概率设置作为遗传源，然后使游戏物体存活时间最长，合并、进化它们的属性，这样就给了后面产生的物体最好的特性。当然，你只需要在大量创建物体时这么做，但思路你已经有了。

模糊逻辑

要介绍的最后一种技术是模糊逻辑，这也可能是最有趣的技术。模糊逻辑与模糊集合论有关。换句话说，模糊逻辑是一种分析数据集（数据集的元素有局部包含物）方法。大多数人习惯于确定逻辑，就是某事包含在一个集合中或不包含在这个集合中。例如，我创建两个集合——儿童，成人。我是属于成人组，而我 3 岁的外甥属于儿童组。这就是确定逻辑。

模糊逻辑则不然，它允许物体不完全包含在一个集合中。例如，我可以说我有 10% 包含在儿童集合中，有 100% 包含在成人集合中。同样，我的外甥可以有 2% 包含在成人集合中，有 100% 包含在儿童集合中。这些都是模糊值。你应该注意到包含在两个集合中的值之和不等于 100%，它可以大于或小于 100%——因为这些数值不是概率。然而，事件或状态在不同的集合中的概率之和仍必须等于 1.0。

模糊逻辑可以通过模糊或错误的数据得出正确的结果。确定逻辑不可能做到这一点。如果缺少一个变量或输入值，确定逻辑就不能够使用了。但模糊逻辑系统在缺少变量的情况下仍然可以工作，就像人脑一样。我的意思是说人可以对一些不明确的事情作出决策。

在 AI 的决策、行为选择领域中运用了模糊逻辑，输出/输入过滤应该很明显。下面我们来看看不同的模糊逻辑方法的运用。

标准集合论

一个标准的集合只是一些物体的汇集。编写一个集合，用大写字母代表集合，然后把集合中的元素放到大括号中间，用逗号隔开元素。任何事物都可以组成一个集合：如姓名、数字、颜色等等。图 12.35 显示了几个标准的集合。

如集合 $A=\{3, 4, 5, 20\}$ ，集合 $B=\{1, 3, 9\}$ 。可以对这些集合进行各种操作，操作符如下：

包含 (\in)——谈起一个集合时，你可能想知道一个对象是否包含在这个集合中。这称为集合包含。因此，“ $3 \in A$ ”也就是“3 是集合 A 的一个元素”是正确的论断，“ $2 \in B$ ”则是错误的。

并集 (\cup)——这个操作符取两个集合中的所有对象，然后将这些对象加入到一个新的集合中，如果同一个对象出现在这两个集合中，那么只有一个加入到新集合中。如：

$A \cup B = \{1, 3, 4, 5, 9, 20\}$ 。

交集 (\cap)——这个操作符只取两个集合中都有的对象。因此, $A \cap B = \{3\}$ 。

子集 (\subset)——有时你可能想知道一个集合是否完全被另一个集合包含。这称之为集合包含, 或子集。因此, $\{1, 3\} \subset B$, 即集合 $\{1, 3\}$ 是 B 的子集, 而 $A \not\subset B$, 即 A 不是 B 的子集。

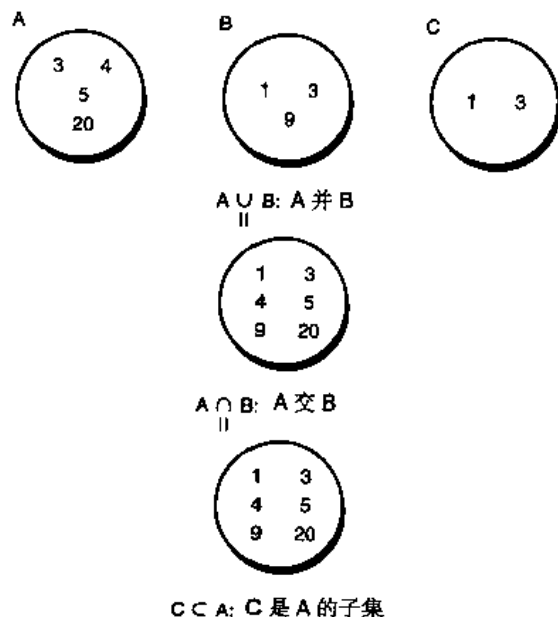


图 12.35 一些简单的集合

数 学

通常用一个斜杠 (/) 或撇 (') 表示取非或求反、倒置等。

这些就是要介绍的集合论。并不复杂, 只有一些术语和符号。其实每个人每天都用到了集合论, 只是不知道而已。标准集合是严格的。元素要么包含在集合中, 要么不包含在集合中。模糊集合论不属于标准集合范畴。

模糊集合论

计算机是精确的机器, 而我们经常用计算机解决一些不严格或模糊的问题——至少我们想这样做。在 70 年代, 计算机科学家开始在软件编程和解决问题中采用一种新的数学技术, 称为模糊逻辑或不确定逻辑。我们所说的模糊逻辑就是模糊集合论及其属性的运用。现在, 我们从刚才学习的标准集合论的角度出发, 来了解模糊集合论。

对于模糊集合论, 你不能再把注意力放在集合中的对象上; 对象是在集合中, 更应该

注意一个特定对象对一特定集合的隶属度。例如，我们创建一个模糊类，称为 Computer Special FX。然后选取一些你喜欢的电影，并估计这些电影在这个模糊类中的适合度。参见表 12.4。

表 12.4 对 Computer Special FX 的隶属度

电影名	对类的隶属度
Antz	100%
Forrest Gump	20%
The Terminator	75%
Aliens	50%
The Matrix	90%

是不是看起来很模糊？虽然 The Matrix 消除了一些计算机生成效果，但 Antz 整个都是计算机生成的，所以必须要公平。然而，你同意这些百分比吗？Antz 是完全计算机生成，有 1 小时 20 分。Forrest Gump 只有 5 分钟的计算机增强图像。让 Gump 占 20% 合适吗？不知道！这就是为什么要使用模糊逻辑的原因。

无论如何，要写出每个模糊隶属度作为一个“{对象，隶属度}”的有序对，因此，这个例子要写成“{ANTZ, 1.00},{Forrest Gump, 0.20},{Terminator, 0.75},{Aliens, 0.50},{The Matrix, 0.9}”。

现在，加上一些提炼，创建一个完整的模糊集合。在大多在数情况下，模糊集合是在一个特定类中的对象集合的隶属度 (DOM) 的有序集。例如，在 Computer Special FX 类中，有一个由隶属度组成的集合： $A=\{1.0,0.20,0.75,0.50,0.90\}$ 。集合中对应每个电影有一个条目——每个都代表每个电影的 DOM (在表 12.4 中列出)。

假设有另一个电影集合，里面是电影本身的隶属度： $B=\{0.2,0.45,0.5,0.9,0.15\}$ 。现在，进行集合操作，并观察结果。在执行集合操作前，有一个中止申请。因为我们所说的模糊集合表示一个对象集的隶属度或向量适合度，很多集合操作要求每个集合中的对象数量必须相同。看了下面的集合操作后，就会清楚了。

模糊并集 (\cup)——两个模糊集合的并集是这两个集合的相应元素中的最大元素的集合。例如，有两个模糊集合：

$A=\{1.0,0.20,0.75,0.50,0.90\}$

$B=\{0.2,0.45,0.5,0.9,0.15\}$

A 和 B 的并集为：

$A \cup B = \{\text{MAX}(1.0,0.2), \text{MAX}(0.20,0.45), \text{MAX}(0.75,0.5), \text{MAX}(0.90,0.15)\}$
 $= \{1.0,0.45,0.75,0.90\}$

模糊交集 (\cap)——两个模糊集合的交集是这两个集合的相应元素中的最小元素的集

合。例如，有两个模糊集合：

$$A = \{1.0, 0.20, 0.75, 0.50, 0.90\}$$

$$B = \{0.2, 0.45, 0.5, 0.9, 0.15\}$$

A 和 B 的交集为：

$$\begin{aligned} A \cap B &= \{\text{MIN}(1.0, 0.2), \text{MIN}(0.20, 0.45), \text{MIN}(0.75, 0.5), \text{MIN}(0.90, 0.15)\} \\ &= \{0.2, 0.20, 0.50, 0.15\} \end{aligned}$$

模糊集合的包含和子集没有多大意义，所以我不介绍了。然而，一个模糊值或模糊集合的求反是有意义的。一个隶属度为 x 的模糊变量的求反为 $(1-x)$ 。

$$A = \{1.0, 0.20, 0.75, 0.50, 0.90\}$$

所以 A 的求反（写作 A' ）为：

$$\begin{aligned} A' &= \{1.0-1.0, 1.0-0.20, 1.0-0.75, 1.0-0.50, 1.0-0.90\} \\ &= \{0, 0.8, 0.25, 0.5, 0.1\} \end{aligned}$$

模糊语言变量和规则

现在，你已经知道怎样引用模糊变量或模糊集，下面我们来看看怎样运用到游戏 AI 中。我们将要创建一个 AI 引擎，创建过程中要运用模糊规则，并将模糊逻辑应用到输入中。这个 AI 引擎可以控制游戏物体的模糊输出或确定输出。如图 12.36 所示。

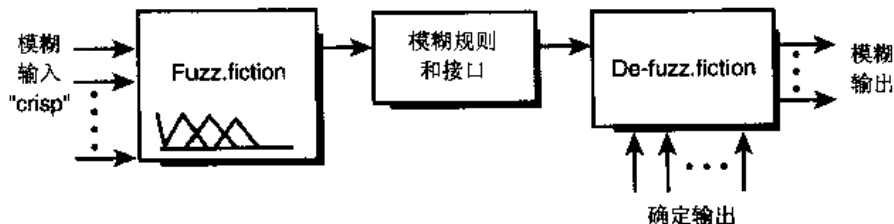


图 12.36 模糊 I/O 系统

使用标准条件逻辑编写可以一些语句，或带有下列形式陈述的树如：

```
if X AND Y then Z
```

或

```
if X OR Y then Z
```

X 和 Y 变量称为前提条件，Z 称为结论。然而，在模糊逻辑中，X 和 Y 是模糊语言变量（缩写为 FLV）。而且，Z 也可以是 FLV 或者是个明确的值。关键是 X 和 Y 代表模糊变量，所以 X 和 Y 都是不确定的。这种形式的模糊命题称为规则，规则最终要经过一些步骤才能估算出来，不能用下面这样的方式估计结果：

```
if EXPLOSION AND DAMAGE then RUN
```

对于模糊逻辑，规则只是最终结果的一部分。模糊化和消除模糊化才是最终结果。

FLV 代表与范围有关的模糊概念。例如，假设用 3 种不同的模糊语言变量把玩家与 AI 物体之间的距离分类。参见图 12.37。图中显示了一个模糊流或模糊面，由 3 个不同的三角区域组成，这 3 个区域的标注为：

NEAR	区域范围 (0~300)
CLOSE	区域范围 (250~700)
FAR	区域范围 (500~1000)

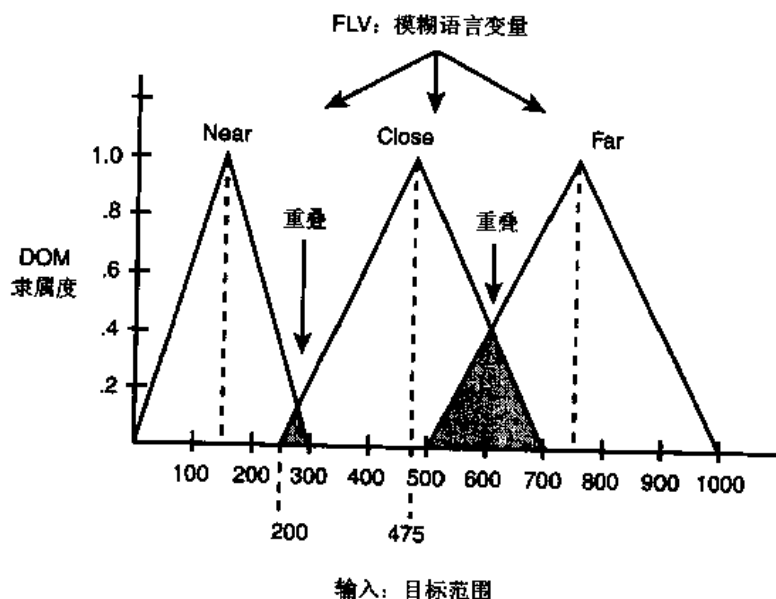


图 12.37 由 FLV 区域组成的模糊流

x 轴为输入变量，范围从 0 到 1000。这称为区域。Y 轴为模糊流的输出，范围从 0.0 到 1.0。对于任意输入值 x_i (在这个例子中表示到玩家的距离)，垂直引出一条线就可以用计算出隶属度 (DOM)，如图 12.38 所示，即垂线与每个模糊语言变量的三角区域的交点处的 Y 值。模糊面上每个三角形代表相应模糊语言变量 (NEAR, CLOSE, FAR) 的影响区域。另外，这些区域有一部分重叠，通常为 10%~50%。这是因为当 NEAR 转换为 CLOSE, CLOSE 转换为 FAR 时，不能立即就转换对应的值。这一部分重叠模拟了情形的模糊。这就是模糊逻辑的思想。

注意

前面的 FSM 例子中，用相似的技术来选择状态 (在本章前面“有逻辑条件的模式”这一部分中)。检测到目标的距离，如果距离达到一定值就会使 FMS 转换状态。但 FMS 中，使用的是确定的值，没有重叠或模糊计算。例子中，从“逃脱”到“攻击”的转换过程中有确定的区域。但是，对于模糊逻辑，这个区域则很不清楚。

我们来概括一下。现在有了基于从游戏引擎，环境中模糊输入的规则。这些规则看起

来像是标准的条件逻辑语句，但必须用模糊逻辑计算，因为这些规则是真正的根据隶属度划分输入的 FLV。

而且，模糊逻辑处理的最终结果可能被转化为不连续的确定值，如“射击”，“跑动”，“静止”，或转化成连续的值如能量级从 0~100。模糊逻辑处理的最终结果也可以用于模糊处理。

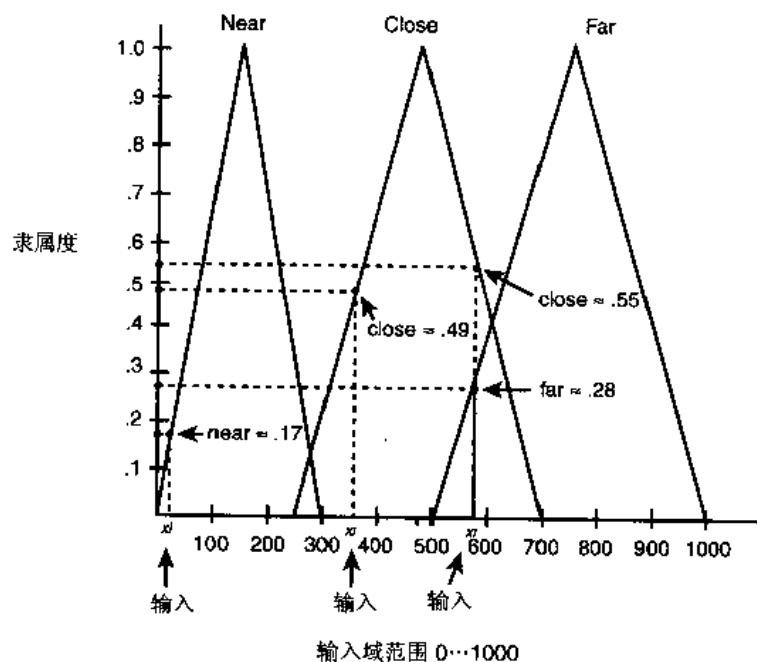


图 12.38 在一个或多个 FLV 中计算范围值隶属度

模糊流和模糊对象

如果模糊逻辑 AI 系统有一些输入值，这些输入值分为一种或多种（一般是多种）模糊逻辑变量 FLV（这表示一些模糊范围），然后计算每个输入值在每个 FLV 范围中的隶属度。通常，在输入值 x_i 范围内，什么是每个模糊逻辑变量 NEAR、CLOSE、FAR 的隶属度？

到现在为止，模糊语言变量是用对称三角形定义的区域。然而，你不仅可以用对称三角形，而且可以用梯形、S 形函数等来定义模糊语言变量。如图 12.39 所示。

大多数情况，使用对称三角形更加方便。如果你想使 FLV 的范围恒等于 1.0，可以使用梯形。总之，计算任何输入值 x_i 在一个特定 FLV 中的隶属度（DOM），只要在 x 轴上取到输入值 x_i ，然后做一条垂线，垂线与三角形的交点的纵坐标值就是 DOM。

用软件计算这个值很容易。现在，我们假设对于每个 FLV 用一个三角形表示，三角形的左右起点为 min_range 、 max_range ，如图 12.40 所示。

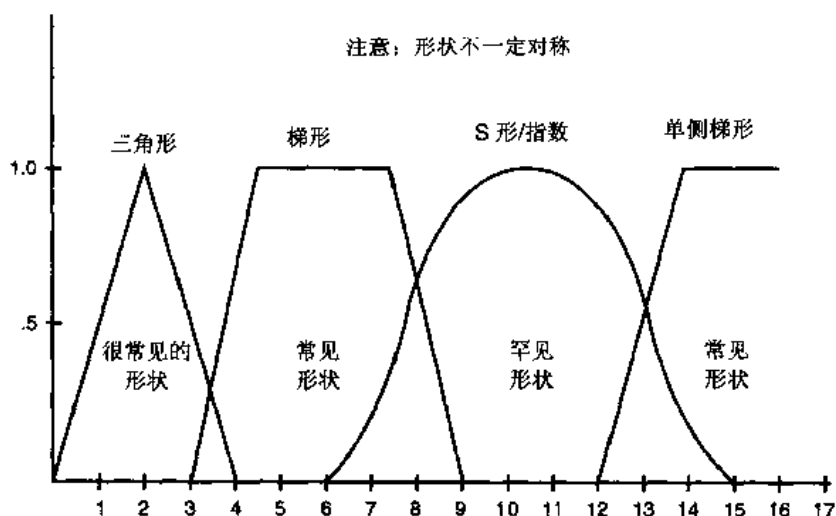


图 12.39 典型的模糊语言变量几何图形

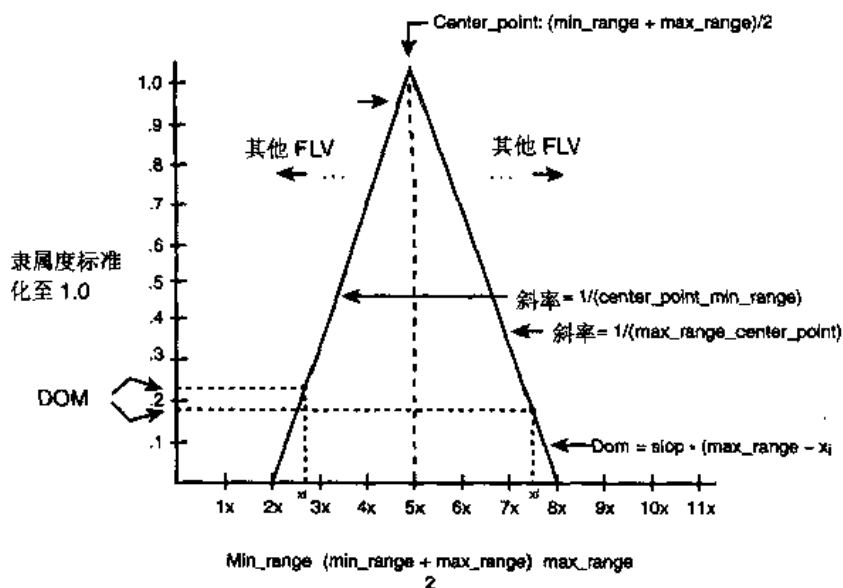


图 12.40 计算每个 FLV 的 DOM

计算任意给定输入值 x_i ，可以用下面这个算法：

```
// first test if the input is in range
if(xi >=min_range && xi <= max_range)
{
// compute intersection with left edge or right
// always assume height of triangle is 1.0
float center_point = (max_range + min_range)/2;
// compute xi to center
if(xi<=center_point)
{
```

```

// compute intersection on left edge
// dy/dx = 0.1/(center - left)
slope = 1.0/(center_point - min_range);
degree_of_membership = (xi - min_range) * slope;
} // end if
else
{
    // compute intersection on right edge
    // dy/dx = 1.0/(center - right)
    slope = 1.0/(center_point - max_range);

    degree_of_membership = (xi - max_range);

    } // end else
} // end if
else // not in range
    degree_of_membership = 0.0;

```

当然，这个算法可以优化，但我主要是想让你看看这个过程。如果使用梯形，就有 3 个可能相交的区域：左边线、右边线、上底线。

大多数情况，至少应该有 3 个模糊语言变量。如果超过 3 个，最好保证模糊语言变量的个数为奇数，这样就会总有一个变量居中。否则，在模糊区域中间就会有一个凹槽。

我们来看一些计算前面的模糊流的隶属度的例子，如图 12.37 所示。基本上，对于任意输入值，作一条垂线，确定垂线与模糊流中的 FLV 的交点，垂线可能与多个 FLV 相交，需要解决这个问题。首先，我们要得到一些 DOM。

假设输入域 $x_i = \{50, 75, 250, 450, 550, 800\}$ 。如图 12.41 所示。

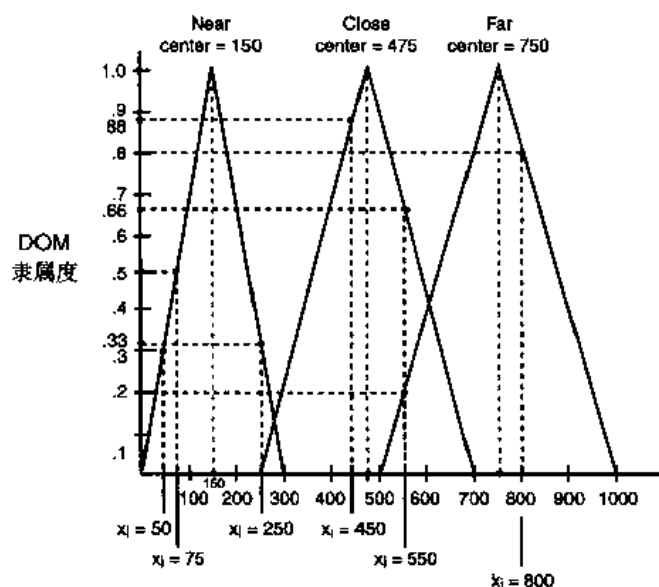


图 12.41 带有输入的模糊流

在这个例子中，每个 FLV (NEAR, CLOSE, FAR) 的隶属度可以用算法计算出来，也可以直接从图中读出。结果列在表 12.5 中。

表 12.5 模糊流区域的隶属度的计算

输入“到目标的距离” x_i	隶属度		
	NEAR	CLOSE	FAR
50	0.33	0.0	0.0
75	0.5	0.0	0.0
250	0.33	0.0	0.0
450	0.0	0.88	0.0
550	0.0	0.66	0.20
800	0.0	0.0	0.80

仔细观察这个表，会发现一些有趣的属性。首先，对于任意输入值，成员结果之和不等于 1.0。记住，这些值是隶属度，而不是概率值，所以是合适的。

有一些输入值 x_i 的 DOM 在一个或两个不同的模糊变量中，也有可能在 3 个不同的模糊变量中（如果把三角形设计得足够大）。选择每个三角形尺寸（范围）的过程称为校准，有时要经过多次校准后才能得到想要的结果。有 3 个以上的 FLV，必须经过多次校准后才能得到很好的效果。

CD 中的 DEMO12_9.CPPIEXE 是创建一个模糊流的例子。它使你可以创建一些模糊语言变量——即一些输入域的种类。然后输入一些数值，就会得到每个输入值的隶属度。这是个控制台应用程序，因此要编译它。通过取每个 DOM 值，并把 DOM 总和分类，实现在每次计算时将成员的打印数据标准化为 1.0。

现在，你已经知道怎样创建一个模糊流。然后在区域里选择一个输入值，计算模糊流中的每个 FLV 的隶属度，建立一个有特定输入值组成的集合。这称为模糊化。

只有当你模糊化两个或两个以上的变量，然后用规则把结果联系起来，并观察结果，模糊逻辑才起到作用。为达到这一步，首先必须将另一个输入模糊化。图 12.42 显示了模糊流的输入能量级。

模糊语言变量如下：

WEAK 区域范围 (0.0~3.0)
 NORMAL 区域范围 (2.0~8.0)
 ENERGIZED 区域范围 (6.0~10.0)

注意到模糊变量的区域范围是从 0 到 10.0，而不是从 0 到 1000 (0 到 1000 可以为玩家变量的范围)。这总的来说可以接受。可以加上 3 个或 3 个以上的 FLV，但 3 个可以使问题对称。处理模糊变量需要构造一个规则库并创建一个模糊伴随矩阵，这就是我们下面将

要讨论的问题。

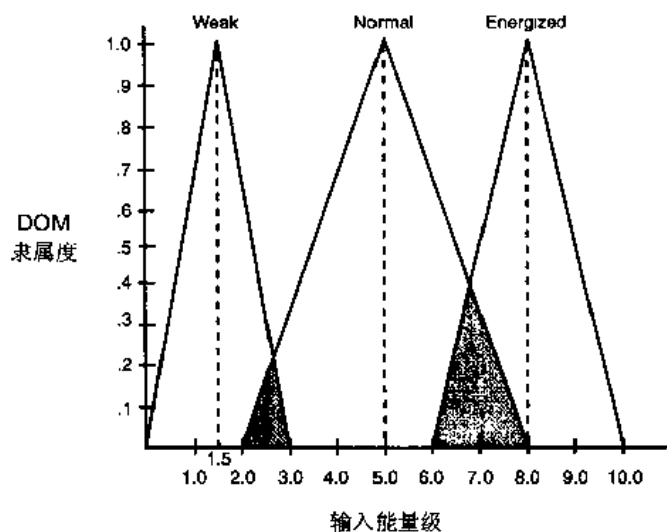


图 12.42 不同能量等级的模糊流

模糊伴随矩阵

模糊伴随矩阵 (FAM) 用于从一个或多个模糊输入及给定的规则库中推导出几个结果，并输出一个模糊值或确定值。如图 12.43 所示。

		输入 X		
		x ₁	x ₂	x ₃
输入 Y	y ₁	If x ₁ and y ₁ then z ₁₁	If x ₂ and y ₂ then z ₂₁	If x ₃ and y ₃ then z ₃₁
	y ₂	If x ₁ and y ₂ then z ₂₁	If x ₂ and y ₂ then z ₂₂	If x ₃ and y ₂ then z ₃₂
	y ₃	If x ₁ and y ₃ then z ₃₁	If x ₂ and y ₃ then z ₃₂	If x ₂ and y ₃ then z ₃₃

图 12.43 运用模糊伴随矩阵

在大多数情况下，FAM 只处理两个模糊变量，因为这样就可以用一个 2D 矩阵排列：一个变量代表一个轴。矩阵中每一项是逻辑命题“if X_i AND Y_i then Z_i ”， X_i 是 x 轴上的模糊逻辑变量， Y_i 是 y 轴上的模糊逻辑变量， Z 是结果——可能是一个模糊变量或是一个确定值。

建立一个 FAM，需要知道规则和每个矩阵项的输出。换句话说，需要建立一个规则库，并选定输出变量是确定的还是线性的。一个确定的输出应该是{“攻击”，“迷路”，“寻找”}，而一个线性输出可能是一个从 0 到 100 的延伸等级。相对来说无论获得那一个都是差不多的；无论那种情况，都必须对 FAM 的输出进行逆模糊化，并得到一个新的输出。

下面你将会看到两个例子，一个是选择一个类的确定单一输出，一个是在一个范围内只输出一个值。这两个例子大部分是一样。首先，我们来看看将计算一个范围作为最终输出的例子：

1. 选择输入，定义 FLV，然后建立模糊流。

这个输入值是由 AI 控制的物体到玩家的距离和自身的能量等级。

输入 X 物体到玩家的距离

输入 Y 物体的能量等级

参见前面的图 12.37 和图 12.42。

2. 建立一个规则库。

规则库只是一些逻辑命题形式（如“if X AND Y then Z ”，“if X OR Y then Z ”）的汇集。在计算 FAM 输出时，规则库发挥着很重要的作用。在模糊集合论中，逻辑 AND 意思是“集合中最小值”，而逻辑 OR 意思是“集合中最大值”。现在开始，我们全部使用 AND，不过在后面部分，我将会介绍怎样使用 OR。

一般来说，如果有两个模糊输入，每个输入有 m 个 FLV，那么模糊伴随矩阵的维数为 $m \times m$ 。由于每一项代表一个逻辑命题，这就意味着需要 9 个规则（ $3 \times 3 = 9$ ）来每一项可能的逻辑组合和输出。

不过，这并不必要。如果只有 4 个规则，在 FAM 中可以将其他的输出设置为 0.0。尽管如此，在这个例子中我们还是使用 9 个规则。对于一个输出，将会用到模糊输出延伸等级，现在来创建一个由下面这些模糊类别组成的模糊变量：

OFF	区域范围 (0~2)
ON HALF	区域范围 (1~8)
ON FULL	区域范围 (6~10)

这些 FLV 的模糊流如图 12.44 所示。

输出可以有多个模糊类别，在这我只用了 3 个。下面是任意规则：

输入 1: 物体到玩家的距离。

NEAR

CLOSE

FAR

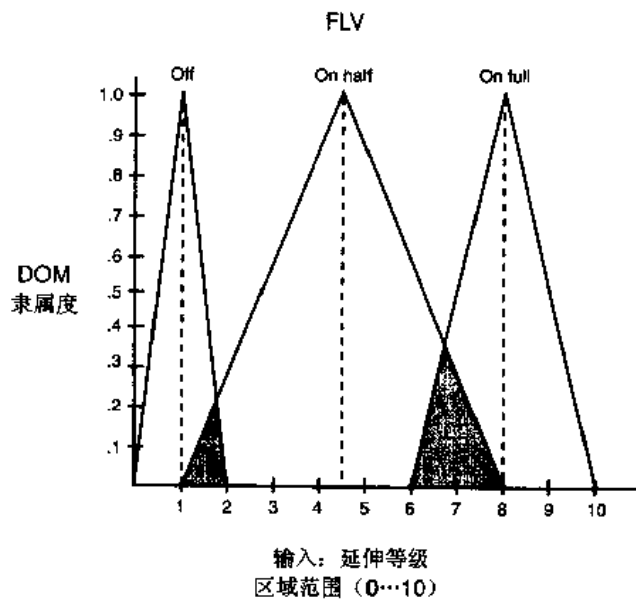


图 12.44 不同延伸等级下的输出模糊流

输入 2: 物体的能量等级。

WEAK
NORMAL
ENERGIZED

输出: 内部导向延伸等级。

OFF
ON HALF
ON FULL

规则: 组合。

```
if NEAR AND WEAK then ON HALF
if NEAR AND NORMAL then ON HALF
if NEAR AND ENERGIZED then ON FULL
if CLOSE AND WEAK then OFF
if CLOSE AND NORMAL then ON HALF
if CLOSE AND ENERGIZED then ON HALF
if FAR AND WEAK then OFF
if FAR AND NORMAL then ON FULL
if FAR AND ENERGIZED then ON FULL
```

实际上，这些规则是启发式的，是从一个“专家”得知在这些情况下 AI 应该要做什么。这些规则看起来有些矛盾，需要仔细研究几分钟！现在，既然有了这些规则，就把这些规则完全带入到模糊伴随矩阵中，如图 12.45 所示。

输入 1=到玩家的距离

		Near	Close	Far
输入 2=自身能量等级	Weak	And On half	And Off	And Off
	Normal	And On half	And On half	And On full
	Energized	And On full	And On half	And On full

图 12.45 包含所有规则的 FAM

用模糊化的输入处理 FAM

处理 FAM 的步骤如下：

1. 从每个模糊变量中获取确定输入值，然后通过计算这些确定输入值在每个 FLV 中的 DOM 来模糊化这些输入值。例如，输入值为：

输入 1 物体到玩家的距离 = 275

输入 2 物体的能量等级 = 6.5

为了模糊化这两个输入，可将它们输入到两个模糊流中，并计算每个输入对各个模糊变量的隶属度。参见图 12.46。

输入 1 = 275，每个 FLV 的隶属度如下：

NEAR 0.16

CLOSE 0.11

FAR 0.0

输入 2 = 6.5，每个 FLV 的隶属度如下：

WEAK 0.0

NORMAL 0.5

ENERGIZED 0.25

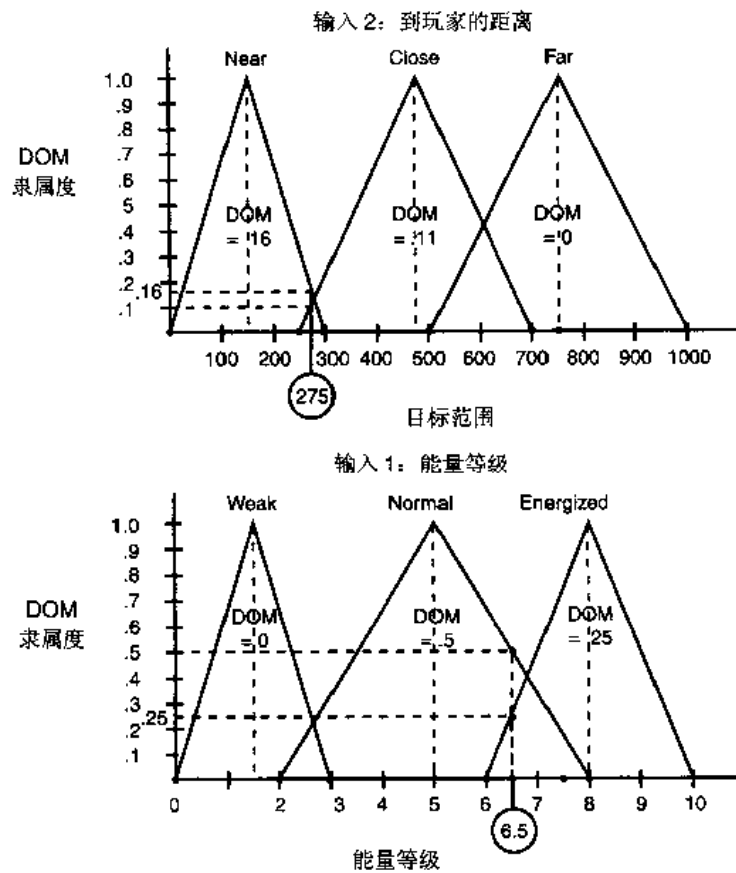


图 12.46 模糊变量带入一些输入值

现在，引用模糊伴随矩阵，然后检查矩阵中每一项的规则，并查看基于前面模糊数值的输出值。当然，FAM 中有些项为 0.0，这是因为有两个 FLV 为 0.0。参见图 12.47，图中描述了所有 FAM 项，非零项左上角用阴影标出。

FAM 的每一项代表一个规则。例如，左上部位的项表示

```
if NEAR AND WEAK then ON HALF
```

要估算这个规则，用 MIN () 规则（即逻辑 AND）对前提条件进行测试。在这个例子中，NEAR = 0.16，WEAK = 0.0，所以：

```
if (0.16) AND (0.0) then ON HALF
```

用 MIN () 进行计算：

$$(0.16) \wedge (0.0) = (0.0)$$

因此，这个规则根本就不会生成。另一方面，我们来看看另一个规则：

输入 2=275

		Near = .16	Close = .11	Far = 0
输入 2=6.5	Weak = 0	0 and .16 = 0 $0 \wedge .16 = 0$ $\min(0, .16) = 0$ Output: On half Value: 0	0 and .11 = 0 $0 \wedge .11 = 0$ $\min(0, .11) = 0$ Output: Off Value: 0	0 and 0 = 0 $0 \wedge 0 = 0$ $\min(0, 0) = 0$ Output: Off Value: 0
	Normal = .5	.5 and .16 = .16 $.5 \wedge .16 = .16$ $\min(.5, .16) = .16$ Output: *On half Value: .16	.5 and .11 = .11 $.5 \wedge .11 = .11$ $\min(.5, .11) = .11$ Output: *On half Value: .11	.5 and 0 = 0 $.5 \wedge 0 = 0$ $\min(.5, 0) = 0$ Output: On full Value: 0
	Energized = .25	.25 and .16 = .16 $.25 \wedge .16 = .16$ $\min(.25, .16) = .16$ Output: *On full Value: .16	.25 and .11 = .11 $.25 \wedge .11 = .11$ $\min(.25, .11) = .11$ Output: *On half Value: .11	.25 and 0 = 0 $.25 \wedge 0 = 0$ $\min(.25, 0) = 0$ Output: On full Value: 0

* - means this rule fire S

图 12.47 包含所有运算和输出值的模糊伴随矩阵

```
if CLOSE AND ENERGIZED then ON HALF
```

意思是:

```
if(0.11) AND (0.25) then ON HALF
```

用 WIN () 计算:

$$(0.11) \mid (0.25) = (0.11)$$

规则 ONHALF 生成一个 0.11, 所以用这个值代替 FAM 中 CLOSE 和 ENERGIZED 的交点处的规则 ON HALF。按照这个步骤做下去, 直到访问完所有 FAM 项。如图 12.47 所示。

最后, 我们要模糊化 FAM。模糊化 FAM 有很多种方法。基本上, 都需要一个最终确定值代表延伸等级 (0.0~10.0)。有两种主要的方法计算出这个值: 可以使用逻辑和或 MAX () 方法求这个值, 也可以使用基于模糊质心的求平均值技术。我们首先来看看 MAX () 方法。

方法 1: MAX 技术

如果你观察 FAM 数据, 就会得到下面这些模糊输出值。

```
OFF                {0.0}
ON HALF            {0.16, 0.11, 0.16}, 和为 0.43
```

ON FULL (0.16)

注意到规则 ON HALF 在 3 个不同的输出中生成不同的值。所以需要决定怎样处理这些值。是用求和，还是求平均值，还是求最大值？这取决于你自己。这个例子中，选择求和： $0.16+0.11+0.16=0.43$ 。

这还是一个模糊值，但观察这些数据，看起来 ON HALF 是最强的成员。所以，可以这样做：

```
output = MAX(OFF , ON HALF , ON FULL)
        = MAX(0.0,0.43,0.16)=0.43
```

使用逻辑和操作符 v:

$(0.0) \vee (0.43) \vee (0.16) = (0.43)$

然后，用(0.43) 乘以输出的范围，如下：

$(0.43) * (10) = (4.3)$

设置延伸度为 (4.3)。

这种方法的惟一问题就是即使所选取的变量是值最大的成员，但变量在模糊区域的整个影响可能会非常小。譬如，40%NORAML 的影响要大于 50%WEAK。更好的方法是把一些值带入到输出模糊流 (OFF, ON HALF, ON FULL) 中，计算区域影响度，然后计算整个的质心，并将这个质心作为最后的输出值。

方法 2: 模糊质心

要找到模糊质心，在输出中取每个 FLV 的模糊值：

OFF	(0.0)
ON HALF	(0.14) {平均值}
ON FULL	(0.16)

把这些值带入到 FLV 图形的 y 轴，并填充每个区域。如图 12.48 所示。

把所有填充区域合并，得到一个新的区域，找出新区域的质心。可以看出有两种合并区域的方法：重叠和添加。重叠会损失一些区域，但比较简单。而添加技术则更为精确。

图 12.48 显示了求解质心的这两种方法。但计算机毕竟不是一张图纸。怎样计算质心呢？

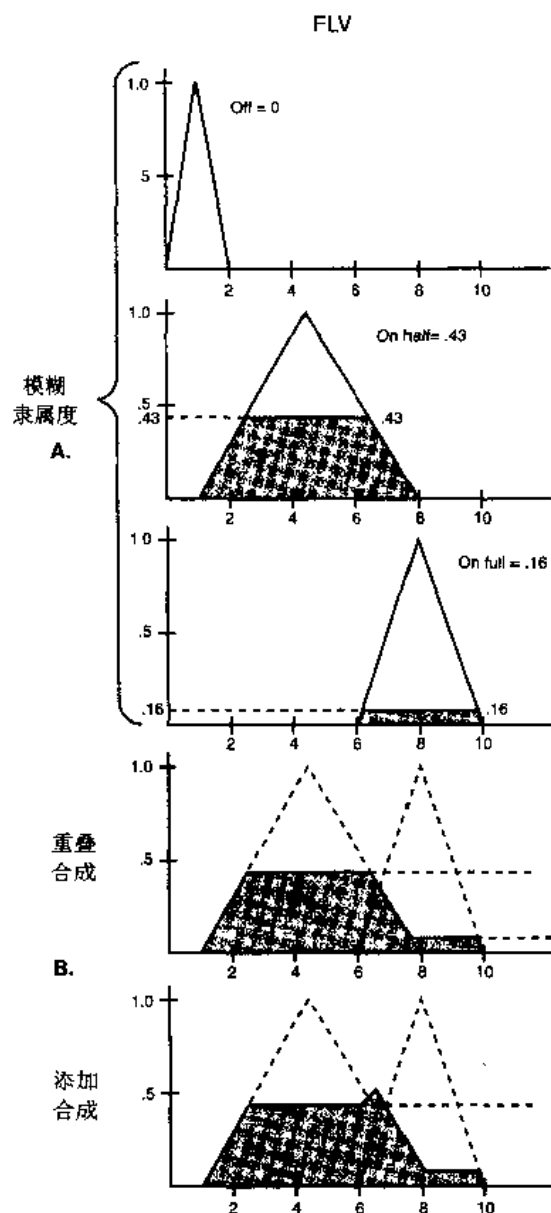


图 12.48 用作图的方法寻找模糊流的区域和质心

想要计算出质心，要进行数值积分（这是微积分术语）。如果想要找到这个模糊区域物体的区域的质心，就需要对每个物体与其对总体的影响度的乘积求和，然后除以总区域：

$$\frac{\sum_{i=1}^{\text{Domain}} d_i \cdot \text{dom}_i}{\sum_{i=1}^{\text{Domain}} \text{dom}_i}$$

di 是区域里的输入值, dom_i 是这个值的隶属度。这用实际例子就特别容易解释。在这个例子中输出区域从 0.0 到 10.0。这代表延伸等级。

需要一个循环变量 di 从 0 循环到 10。在每次循环的间隔, 就要计算这个特定 di 的在混合区域中的隶属度。如图 12.49 所示。

由于每个三角形都被初始值截去一定的高度, 所以应该用梯形来计算隶属度, 而不是用三角形:

```
OFF          (0.0)
ON HALF      (0.14)
ON FULL      (0.16)
```

下面是伪代码:

```
sum          =0.0;
total_area   =0.0;
for (int di=0;di<=10; di++)
{
    // compute next degree of membership and add to
    // total area
    total_area = total_area + degree_of_membership(di);
    //add next contribution of the shape at position di
    sum =sum + di *degree_of_membership(di);
} // end if
// finally compute centroid
centroid = sum/total_area;
```

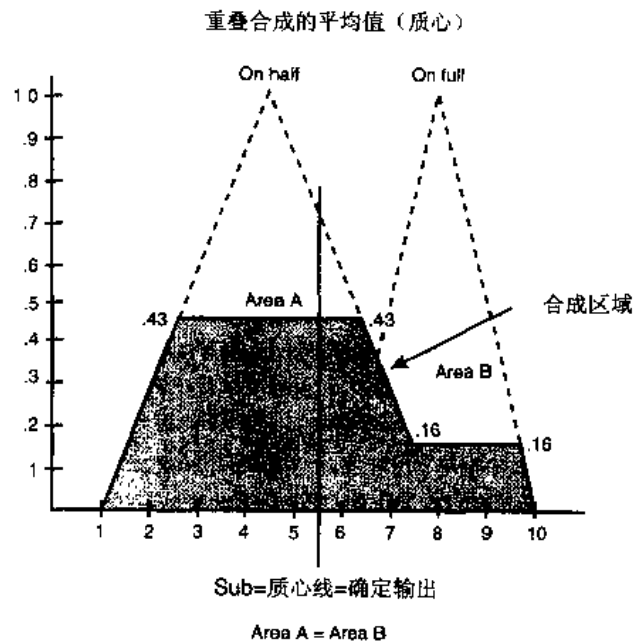


图 12.49 从模糊质心中计算出最终确定输出

函数 `degree_of_membership()` 取初始值 (0~10)，然后将这些值带入到混合输出模糊流中——将下面这些模糊值带入输出变量中，并找到每个变量的区域影响度，这样就可以得到混合输出模糊流。

```
OFF          (0.0)
ON HALF     (0.14)
ON FULL     (0.16)
```

你可以看出使用 `MAX()` 方法的确要容易得多，而且大多数情况下，它的精度和质心法相差不大。

至于如何使最终输出值是确定值而不是线性值，这非常简单。只要使用 `MAX()` 方法，并将输出分类就可以了。或你可以选择输出区域为 0, 1, 2, 3, 4，刚好有 5 个确定输出命令。

模糊逻辑概述

模糊逻辑的思想很单一，但模糊逻辑的实际实现却是详细的。CD 没有关于模糊逻辑实际实现的演示，不过你可以在 Internet 看到相关的例子。网上有很多商业化的模糊逻辑实验程序很有价值，你可以去看看。

在游戏中创建真正的 AI

这一节是关于基本 AI 技术在游戏中的应用。在前面我介绍了一些 AI 技术引你入门，但你现在可能还不能完全确定选用那种技术和怎样混合使用这些技术建立新的模型。下面是一些基本指导原则：

- 创建有简单行为的物体，如岩石、导弹等，使用单一确定的 AI 技术。
- 如果创建的物体比较灵活，但只是环境的一部分而不是主体部分（如飞来飞去的鸟，偶然飞行一下的宇宙飞船），这需要将确定的 AI 技术与一些模式和随机性相结合。
- 创建对玩家有影响的重要游戏物体，需要将 FSM 和一些其他的支持技术结合起来使用。然而，有些物体不需要像其他物体那样灵活，这些物体不需要有与个性结合的存储学习的概率分布。
- 最后，游戏中的主要物体应该十分灵活。你应该把所有技术结合起来。AI 应该是状态驱动的，有很多条件逻辑、概率、存储来控制状态转换。另外，如果必要的转换的条件成立，AI 应该能够从一种状态转换到另一种状态（即使一种状态还没有结束）。

你不必把精力集中在随机运动的石头上，而应该放在与玩家对抗的坦克上。一个好的模型是一个有能够根据条件和概率选择状态的 AI。状态可以模拟一定数量的行为，通常为 5~10 个。可以用一个存储来跟踪游戏中的主要物体，这样可以作出更好的决策。同样，在许多决策中加入一些随机数，可以增加 AI 的不确定性。

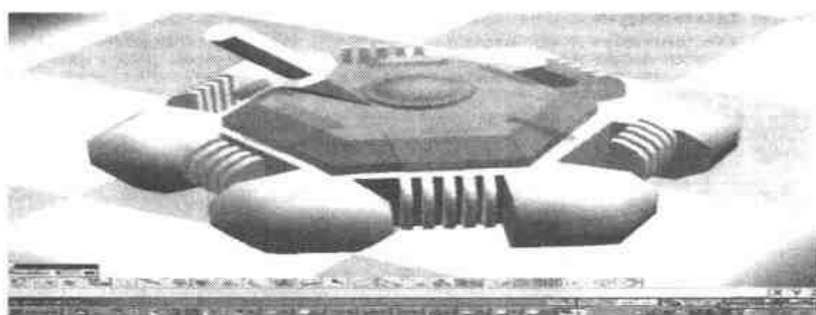
可以将脚本模式用以创建复杂思想。同样，可以在模式中增加随机事件。例如，如果 AI 进入一个模式状态，选择一个圆形，这是正确的。但是，有时可能选择一个蛋形！问题是，人无完人，有时总要有些错误。这个性质在游戏 AI 中非常重要。

最后，一个非常复杂的系统可以从非常简单组成部分发展起来。换句话说，即使 AI 对于单个物体来说可能并不复杂，但物体的互相影响会建立一个紧急行为系统，这个系统看起来要超过设计时想要达到的效果。只要看看我们自己的大脑，这里每个细胞都有很强的自我意识，这对于处理物体间的一些共享或混合信息是很重要的，例如物体靠得十分接近或相隔距离一定。这有助于在模拟中共享信息。

小 结

本章是不是使你大开眼界。本章涉及到了很多领域，也有许多看起来很古怪的知识。不知这一章是否使你开始对人工智能感兴趣了。无论如何，现在你应该很好掌握了 AI。在这里不但介绍了成熟的技术，还有些特定的技术。在这一章中我们讨论了确定性算法、决策树、计划、脚本语言、神经网络、遗传算法、模糊逻辑。不夸张地说，你应该是半个 AI 专家了。

13



基本物理建模

70、80 年代时，视频游戏中还没有大量包含物理学在内。大部分游戏内容为枪战、侦探-破坏游戏、探险游戏等等。然而，进入 90 年代和“3D 时代”后，物理建模就变得越来越重要了。你不能只是简简单单地使游戏中的对象按照一条不现实的路径移动，对象的移动路径至少应该大致是与现实相符。这一章包含了基本的不基于微积分的物理建模。然后，第二册覆盖更多的内容，基于微积分的 2D 和 3D 模型。以下是这一章的目录：

- 物理基本定律
- 万有引力
- 摩擦力
- 碰撞响应
- 正向运动学
- 微粒系统
- 游戏关键

大多数使用物理模型的模拟和游戏中使用的模拟都是基于标准牛顿物理学的模型。标准牛顿物理学在合理规定的尺寸和质量内（即速度比光速小得多，物体比一个单一的原子大得多，但比一个星系小得多），对于运动和物体相当适用。然而，即使建模实际用基本牛顿物理，也会占用计算机大量能量，一个简单的模拟如下雨或撞球台（如果能够正确模拟的话）将会需要用 Pentium III 以上的处理器。

尽管如此，我们却在从 Apple II 到 PC 机都看见了下雨及撞球游戏。这些是怎样编写出来的呢？这些游戏的编写者了解物理，在此基础上建模，并在系统预算资源之内编程，他们创建的模型与做游戏的人在现实生活中的所期待的十分接近。程序由大量的技巧、最优化、

建模系统的假设和简化组成。例如：计算两个球体碰撞后的结果要比计算出两个不规则行星碰撞后的结果容易得多。所以，编程者可能会近似的把游戏中的行星用简单的球体代替（只要物理计算可行）。

在最新的游戏中，物理学需要占用很大的篇幅，这是因为其中不仅仅包括物理，而且还包含需要学习的数学，所以我将只介绍一些最基本的模型。通过这些模型，在你的第一个 2D/3D 游戏中，你将可以对所需要的一切建模。我所介绍的这些物理知识并不比高中物理甚至初中物理多多少。

物理学基本定律

现在我们开始接触物理，主要包括基本的物理概念，时间、空间及物质的属性。这些基本概念将有助于了解下面一些更为先进的主题。

警告



关于量子或宇宙等级，我所说的并不完全正确。但是，我所介绍的在我们所讨论的范围内却是很正确的。另外，我将倾向使用公制。因为英制已有 200 年的历史了，而且现在只有美国人使用它。学术界和其余的人都使用公制。严格说：12 英寸等于 1 英尺，3 英尺等于 1 码，2 码等于 1 英里，5,280 英尺等于 1 英里。

质量 (m)

所有物质都有质量。质量是有多少物质或有多少实际原子质量单位的一个量度。质量与重量无关。许多人混淆了质量和重量。如：千克 (kg) 是公制质量量度，意思是一个物体有多少质量。磅是力的量度，放宽一点来说也可以是重量（质量在重力场中）的量度。重量或力的量度在英制中是磅，在公制中是牛顿 (N)。物质就其本身而论是没有重量的；它只有在重力场中才会产生我们所说的重量。因此，质量是一个比重量（在不同星球上是不同的）更为纯粹的概念。

在游戏中，质量的概念只是一个抽象的相对数量（大多数情况下）。如：我可能设定宇宙飞船等于 100 质量单位，设定小行星等于 10000 质量单位。但除非我在做一个真实的物理模拟，我可以使使用千克作为单位，可这些不是真正的物质。我所需要知道的是一个有 100 质量单位的物体所含有的物质是 50 质量单位的物体所含有的物质的两倍。当介绍力和重力时，我会再提到质量的。质量是度量一个物体由多少物质组成的单位，在公制中使用千克。

质量同样可以看作是度量物体在速度改变时的遇到的阻力的单位——牛顿第一定律。基本上，牛顿第一定律为：一个物体静止时保持静止，一个物体运动时保持运动（以一恒定的速率）直到有一外力作用于物体。

时间 (t)

时间是一个抽象的概念。很难解释时间这个概念而在解释中不用到时间本身。幸运的是，每个人都知道时间是什么，所以我也就不准备研究这，但我想说说时间和游戏中的时间是怎样联系的。

在现实生活中，时间通常是用秒、分、小时等来度量的。或如果你需要精确一点，时间的单位可以是毫秒 (ms, 10^{-3} 秒)、微秒 (μs , 10^{-6} 秒)、纳秒 (10^{-9})、皮秒 (10^{-12}) 等。然而，在视频游戏 (绝大多数游戏) 中，并没有和实际时间有什么很近的联系。帧频与实际时间和秒相比在算法中更为常用，建立时间模型游戏的算法的设计一般都会使用帧频。例如，大多数游戏把一帧当作一虚拟秒，或换句话说，一帧是能够计量的最小时间单位。因此，在大多数时间内，你不会在游戏中和物理模型中使用实际的秒，大都采用虚拟秒，虚拟秒基于作为基本时间等级的单一帧。

另一方面，如果你想创建一个真正完善的 3D 游戏，那么你可能需要用到实际时间。游戏中的所有算法都根据实际时间及固定的帧频，这些算法调整物体的运动。所以一个坦克能够按每秒 100 英尺的速度移动，即使帧频慢到每秒 2 帧或帧频在每秒 60 帧时匀可实现。在这个精度下，建模时间是十分复杂的，但如果你想得到独立于帧频变化的超现实运动和物理事件，那建模时间就是必须的。无论怎样，我们在例子中用秒度量时间，或用仅表示单一帧的虚拟秒度量时间。

位置 (s)

每个物体在 3D 空间有一个 (x, y, z) 位置，或在 2D 空间中有一个 (x, y) 位置，或在 1D 空间 (线性空间) 中有一个 x 位置 (有时当作 s)。图 13.1 显示了所有这些空间位置的例子。然而，有时即使你知道物体在哪，但却不清楚它的位置。例如，如果你必须找

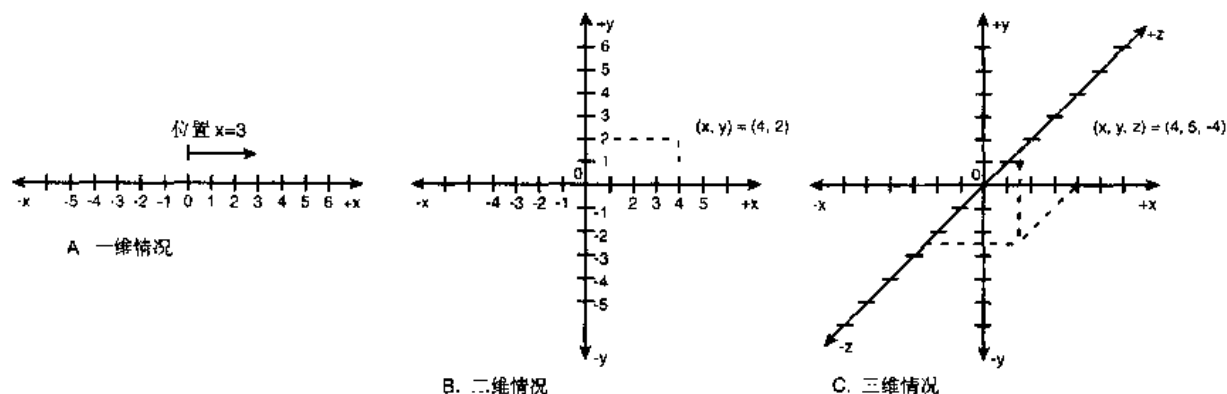


图 13.1 位置的概念

出一个点来定位一个球的位置，那么你可能选择这个球的中心，如图 13.2 所示。但如果是一个锤子？锤子是不规则的形状，所以大多数物理学家将用它的质心或平衡点来定位它的位置，如图 13.3 所示。

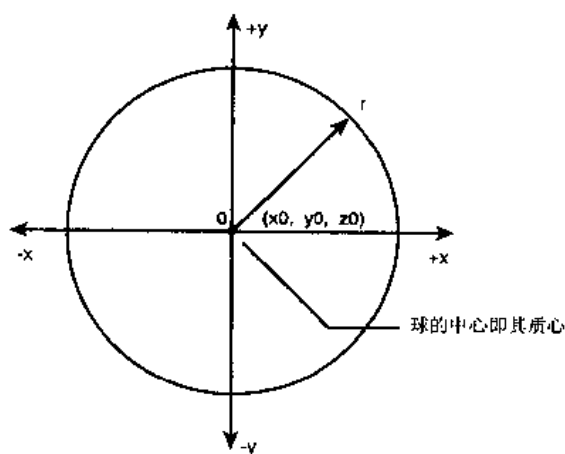


图 13.2 选择一个中心

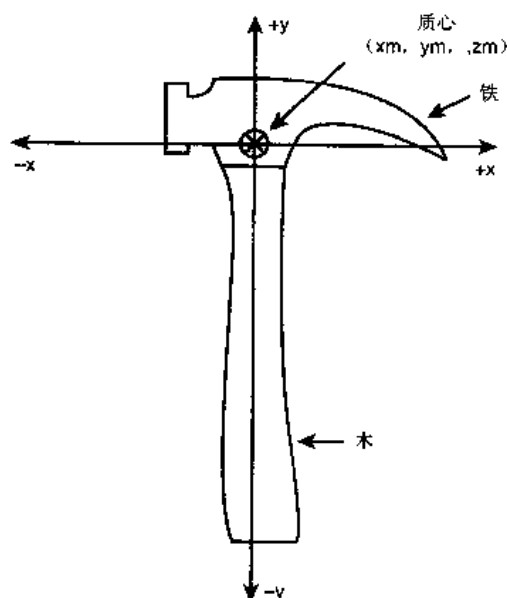


图 13.3 选择不规则物体的中心

位置概念和位置的物理上正确定位，游戏中通常并不是很严格。大多数游戏编程者放置一个边界框、圆或球围绕着所有游戏物体如图 13.4 所示，仅仅使用边界实体的中心作为物体的中心。这用于大多数游戏中，游戏中物体质量的大部分定位在物体的中心，但如果情况并不是这样，那么所有使用这个模拟中心进行的物理计算都会出错。

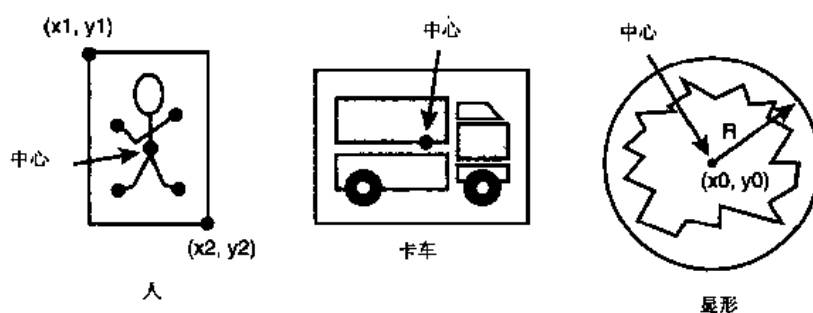


图 13.4 不规则轮廓形状

解决这个问题的惟一方法就是把物体的虚拟质量考虑进去，以找出一个更好的中心。例如，你可以编一个算法，这个算法扫描组成物体的像素，一个区域里像素越多，这个区域就会被认为质量越大。或如果物体是一个多边形的物体，那么你可以给每个顶点附上重量，然后计算出物体质量的真正中心。假设有 n 个顶点，每个顶点的位置标为 (x_i, y_i) ，质量为 m_i ，那么质心为：

$$x_c = \frac{\sum_{i=0}^n x_i m_i}{\sum_{i=0}^n m_i}$$

$$y_c = \frac{\sum_{i=0}^n y_i m_i}{\sum_{i=0}^n m_i}$$

数 学

$\sum f_i$ 意思是“求和”——对于每个 i 值，都加上相应 f_i 。

速率 (v)

速率是一个物体的速度的瞬时值，度量单位通常为米每秒 (m/s)，汽车的速率有时用英里每小时，即 mph。不管你用那个单位，位置随时间改变时，速率也会改变。一维中的速率数学公式为：

$$\text{速率} = v = ds/dt$$

换句话说，是位置（ ds ）相对于时间（ dt ）的瞬时变化。例如，你在公路上行驶，你行驶 100 英里每小时。那么你的平均速率为：

$$v=ds/dt=100 \text{ 英里/1 小时}=100\text{mph}$$

在视频游戏中，速率的概念用到时间单位，但同时单位是任意的也是相对的。例如，在一些我写过的例子中，我总是在 x 轴或 y 轴上以 4 个单位每帧的速度移动物体。所用到的代码大致如下：

```
x_position=x_position+x_velocity;
y_position=y_position+y_velocity;
```

这样就转化成 4 像素/帧。但帧不是时间，是吧？实际上，帧只是帧频保持恒定的单位。在 30fps（每秒帧数）中，30fps 等于一秒每帧的 1/30，4 像素每帧转化为：

$$\begin{aligned}\text{虚拟速度} &= 4 \text{ 像素}/(1/30) \text{ 秒} \\ &= 120 \text{ 像素每秒}\end{aligned}$$

因此，在我们的游戏中的物体移动速率是以像素/秒为单位的。如果你有空闲时间，那么你可以来估算一下你的游戏世界中的一个像素有多少虚拟米，计算一下在电脑空间里的米/秒。无论哪种情况，如果你知道了速率，在任意给定的时间或帧，就可以测量物体的确切位置。例如，如果一个物体现在的位置是 x_0 ，它以 4 像素/帧 的速率移动，30 帧过后，物体的新位置为：

$$\text{新位置} = x_0 + 4 \times 30 = x_0 + 120 \text{ 像素}$$

这导出了我们的第一个重要的运动公式：

新位置=旧位置+速率×时间

$$=x_1=x_0+vt$$

这个公式的定义是：一个物体在位置 x_0 开始以 v 的速率移动，运动 t 秒后，所在的新位置等于它的开始位置加上速率乘上运动时间。看看图 13.5 你就会更清楚一些。我做了一个恒定速率的演示——DEMO13_1.CPP1EXE，演示中将一个物体从屏幕的左端移到右端。

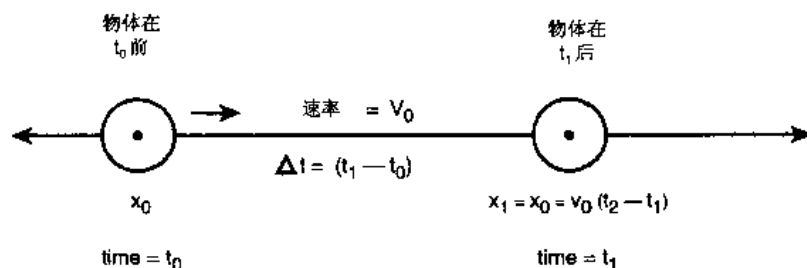


图 13.5 匀速运动

技巧



我每次在汽车上告诉我的朋友需要多长时间就可以驶出坡道，或到达目的地。朋友都十分惊讶。其实这很简单；只要看看速度，然后使用基本知识：如果速度为 60mph，那么每分钟就会行驶 1 英里。所以如果汽车的速度为 60，坡道的长度为 2 英里，那么只要 2 分钟就可以驶出坡道。另一方面，如果汽车的速度为 60，坡道的长度为 3.5 英里，那么要 3 分 30 秒才能驶出坡道。如果行驶速度不为 60mph，那么加上 30 或减去 30 使之与实际速度最为接近。例如，行驶速度为 80，那么可以用 90mph (1.5 米每秒) 计算。然后再调整一下结果。

加速度 (a)

加速度与速率相似，但加速度是度量速率改变快慢的单位，而不是速率本身。看看图 13.6；图中描述了一个物体以恒定速率运动，一个物体以不同的速率运动。以恒定速率运行的物体有一条平行线（斜率为 0），因为物体的速率是时间的一个函数，但有加速度的物体的斜率不为 0，因为它的速率作为时间的函数是改变的。

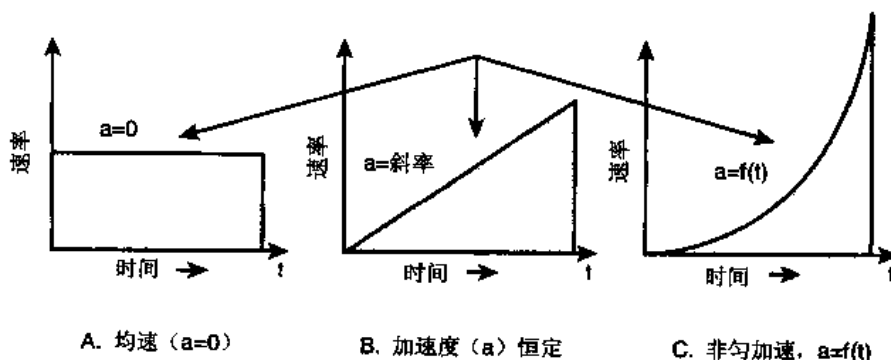


图 13.6 速率—加速度

图 13.6 描述了恒定的加速度，图中也有非恒定加速度。图中 C 部分就是非恒定加速度，图中的线是一条曲线。压缩汽车的加速器就会给你非恒定加速度的感觉，从悬崖上跳下会给你恒定加速度的感觉。从数学上来说，加速度是速率相对于时间改变的比率：

$$\text{加速度} = a = dv/dt$$

加速度的单位有些古怪。因为速率已经是距离每秒的单位，而加速度是距离每秒*秒的单位，或公制中表示为 m/s^2 。加速度是每秒中速率的改变量。而且，牛顿第二定律与速率、时间、加速度有关。第二定律为：在某一时间 t ，速率等于最初速率加上加速时间乘以加速度的积：

$$\text{新速率} = \text{最初速率} + \text{加速度} \times \text{时间}$$

$$= v_1 = v_0 + at$$

加速度是一个相当简单的概念，能够用多种方式建模，我们来看一个简单的例子。

假设一个物体在 (0, 0) 处，它的初始速率为 0。如果用一个恒定的速率 2m/s 给物体

加速，可以仅仅通过用加速度加上最近速率，就会得到新速率，如表 13.1 所示。

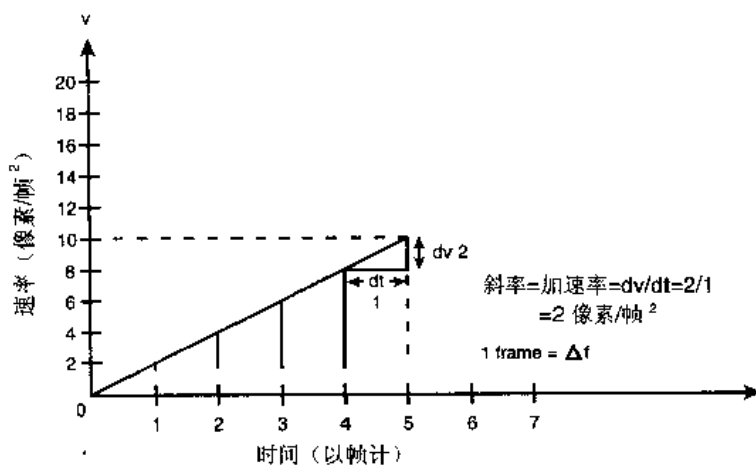
表 13.1 速率为加速度为 2m/s^2 的时间函数

时间 ($t=\text{s}$)	加速度 ($a=\text{m/s}^2$)	速率 ($v=\text{m/s}$)
0	2	0
1	2	2
2	2	4
3	2	6
4	2	8
5	2	10

把表中的数据考虑进去，下一步是找出位置、速率、加速度、时间之间的联系。不幸的是，计算比较复杂，所以我仅仅给出在某一时间 t 时位置的表达式：

$$x_t = x_0 + v_0 t + \frac{1}{2} a t^2$$

这个方程式表示一个物体在某一时间 t 时的位置，等于它的初始位置加上初始速度乘以时间的积再加上加速度与时间平方的积的一半。让我们看看如何把这个方程式运用到游戏中的像素和帧。参见图 13.7。



K 的位置:

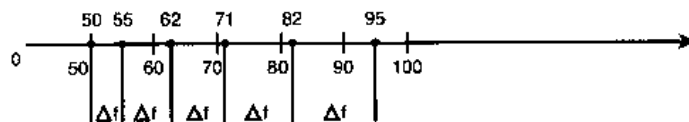


图 13.7 加速度为像素/帧²

假设这些初始条件：物体是在 $x=50$ 像素的位置上，初始速率为 4 像素/帧，加速度为 2

像素/帧²。最后，假设这些都处于 0 帧下。在 C/C++ 中，确定任意时间物体的位置，用以下公式：

$$x = 50 + 4t + (0.5) \times 2 \times t \times t;$$

公式中 t 为帧数。表 13.2 列出了当 $t=0, 1, 2 \dots 5$ 时一些相应的值。

表 13.2 物体以恒定的加速度运动

时间/帧 (t)	位置 (x)	增量 (Δx) $=x_T - x_{T-1}$
0	50	0
1	$50 + 4 \times 1 + (0.5) \times 2 \times 1^2 = 55$	5
2	$50 + 4 \times 2 + (0.5) \times 2 \times 2^2 = 62$	7
3	$50 + 4 \times 3 + (0.5) \times 2 \times 3^2 = 71$	9
4	$50 + 4 \times 4 + (0.5) \times 2 \times 4^2 = 82$	11
5	$50 + 4 \times 5 + (0.5) \times 2 \times 5^2 = 95$	13

表 13.2 中有一些有趣的数据，但最有趣的数据可能是每一时间帧位置的改变量是恒定的，并等于 2。这不是意味着物体每帧移动 2 像素，它表示每帧中运动的变化量变大或增加 2 像素。因此在第一帧时物体移动 5 像素，那么下一帧时物体运动 7 个像素，然后 9, 11, 13 等等。在每个运动变化量之间的增量等于 2 像素，这个增量也就是加速度！

下一步是用 C/C++ 建立加速度模型。基本上，窍门在于：设置一个加速度常量，然后在每一帧中把加速度加上速率。这种方法可以不用前面讲过的长公式——只须用给定的速率转化你的物体。如下：

```
int acceleration=2;// 2 pixels per frame
    velocity    =0;// start velocity off at 1
    x           =0;// start x position of at 0 also
//_
// then you would execute this code each
// cycle to move your object
// with a constant acceleration;
// update velocity
velocity+=acceleration;
// update position
x+=velocity;
```

注 意

当然这个例子是一维的，你只要加上一个 y 坐标就可以升级到二维（ y 方向的速率和加速度自己随意设定）。

为了观察运动中的加速度，我制作了一个演示，名为 DEMO13_2.CPP1EXE。在演示中你可以发射一个导弹，这枚导弹在向前运动时加速。按空格键发射导弹，上下方向键可以

用来增加或减少加速度。A 键用来切换加速度开或关。看看导弹在不同的加速度下的运动运动状况，以及加速度怎样给导弹有“质量”的感觉。

力 (F)

力是物理学中最重要的概念之一。图 13.8 描述了一种力的表达方式。如果一个质量为 m 的物体放在一张桌子上，那么施加在物体上的重力朝地心方向，加速度为 $a=g$ （重力）。重力给质量 m 一个重量，如果你试图提起物体，你会觉得腰有些疼。

牛顿第二定律中力、质量、加速度的相互关系是：

$$F=ma$$

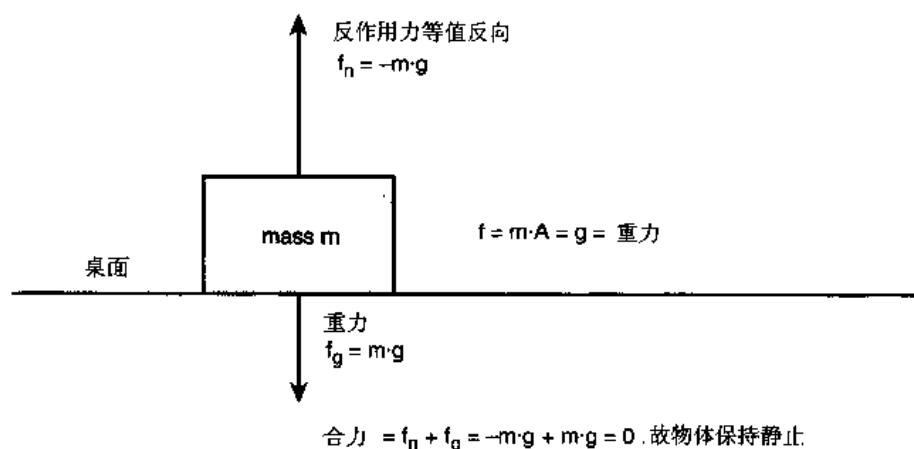


图 13.8 力和重量

换句话说，施加在物体上的力等于质量乘以物体的加速度，重新整理公式为：

$$a=F/m$$

这个公式表示一个物体的加速度在数量上等于你施加在物体上的力除以物体的质量。现在，我们来谈谈测量单位。力等于质量乘以加速度或千克乘以 m/s^2 （ m 代表米，而不是质量）。因此，力的一个可能的单位是：

$$F=kg \cdot m/s^2 \text{ —— 千克} \cdot \text{米每秒平方}$$

这有点长，所以就把这称之为 1 牛顿 (N)。例如，假定一个物体质量 m 为 $100kg$ ，加速度为 $2m/s^2$ 。物体所受的力为 $F=ma=100kg \cdot 2m/s^2=200N$ 。

这可以使你对牛顿有个感性的认识。1 千克的质量大概等于 220 磅的力，而 $1m/s^2$ 的加速运动看上去也不错。

在视频游戏中，有很多原因会用到力的概念，但只要记住以下几个就可以了：

你想对物体施加一个人工力，如爆炸，并计算产生的加速度。

两个物体发生碰撞，你想计算出各个物体所受到的力。

一个游戏武器只有一个力，但它可以发射出各种虚拟质量不同的炮弹，你想知道炮弹

发射时的加速度。

多维空间中的力

当然，力可以作用在所有的三维空间上，而不仅仅只是作用在一条直线上。例如，图 13.9 描述了一个在二维平面中，有三个力作用在一个微粒上。微粒所受到的合力为作用在微粒上所有力的总和。然而，在这种情况下，由于力是矢量，所以不能只是简单地把数量相加。尽管如此，矢量可以被分解成 x 、 y 和 z 分量，这样作用在每个轴上的力可以计算出来。结果为作用在这个微粒上力的和。

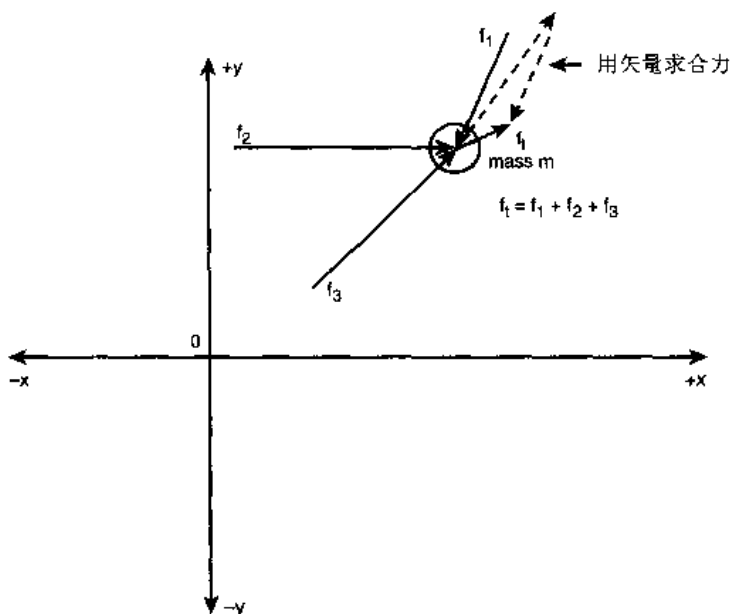


图 13.9 在二维空间中，作用在一个微粒上的力

图 13.9 所示的例子中，有三个力： F_1 、 F_2 和 F_3 ，最终力 $F_{\text{final}} = \langle f_x, f_y \rangle$ 是这些力的总和：

$$f_x = f_{1x} + f_{2x} + f_{3x}$$

$$f_y = f_{1y} + f_{2y} + f_{3y}$$

代入图中的值，你会得到：

$$f_x = (x + x + x) = x \cdot x$$

$$f_y = (y + y + y) = y \cdot y$$

记住以上的内容，就会很容易的推导出在一个物体上的最终力通常为力的矢量和，或数学表示为：

$$F_{\text{final}} = F_1 + F_2 + \dots + F_n$$

这每个力 F_i 可以有 1、2 或 3 个分量，也就是说，每个矢量可以为 1D、2D 或 3D。

动量 (P)

动量是难以口头定义的量之一，基本上它是物体在运动时有的属性。动量是用来度量一个物体的速度和质量的单位。动量是作为一个物体的质量和速率的产物来定义的：

$$P=mv$$

度量单位是 $\text{kg} \cdot \text{m/s}$ ，千克·米每秒。以下的公式得出动量和力的相互关系：

$$F=ma$$

将 P 代入该式

$$F=Pa/v$$

但， $a=dv/dt$ ，因此

$$F = \frac{p \cdot dv/dt}{v} = \frac{d(p) \cdot v}{dt \cdot v} = dp/dt$$

或用文字表示，力为每单位时间内动量的变化率。很有趣，这就意味着，如果物体的动量发生变化，那么就必须有一个力作用在物体上。这有一个无可争辩的论点，一个豌豆可以有和一辆火车一样的动量。一个豌豆的质量可能为 0.001kg ，火车的质量为 1000000kg 。但如果火车的速率为 1m/s ，豌豆的速率为 1000000000m/s （是最快的豌豆），那么豌豆的动量将会比火车的动量大：

$$m_{\text{pea}} \cdot v_{\text{pea}} = 0.001\text{kg} \times 1000000000\text{m/s} = 1000.000\text{kg} \cdot \text{m/s}$$

$$m_{\text{train}} \cdot v_{\text{train}} = 1000000\text{kg} \times 1\text{m/s} = 1000.000\text{kg} \cdot \text{m/s}$$

因此，如果无论哪个物体突然停止，例如撞上什么，这个物体将会感觉到所有的力！这就是为什么在摩托车上一个蜜蜂撞上你都是非常危险的。蜜蜂质量很小，但撞到你的蜜蜂的相对速度非常大。最后得到的动量很大，能够把一个 200 磅的人撞下车。

注意



我骑着 FZR600，速度大约为 155mph，这时有一只蜜蜂撞到我的护目镜上。它不仅仅是使我的护目镜破裂，而且我会感觉像是有人将一个篮球扔向我！

下一步我们学习动量守恒和动量传递。

线性动量的物理性质：守恒和传递

现在你对动量有了一个认识，我们简要地谈谈当一个物体撞上另一个物体时有些什么物理性质。稍后，我会更深入的探讨一下真正的碰撞反应，但现在只是简单地谈谈。

记得在 DOOM 游戏中，当你射中一个桶时，子弹爆炸后会使得桶及这个区域内的敌人移动和/或爆炸。这不是和将一个敌人砸向墙的效果一样吗！这就是动量传递。

通常，如果两个物体相撞，有两种可能：完全弹性碰撞和不完全弹性碰撞。如图 13.10 所示，一个球以速率 v_i 撞向墙壁，如果弹回后它的速率还是 v_i ，动量守恒。所以，这个碰撞是完全弹性的。然而，在现实生活中，很少有这样的情况发生。绝大多数碰撞不是弹性的，至少不是完全弹性的。当碰撞为不完全弹性时，就有些能量转化为热量，或用来变形等。因此，物体碰撞后的动量小于碰撞前的动量。

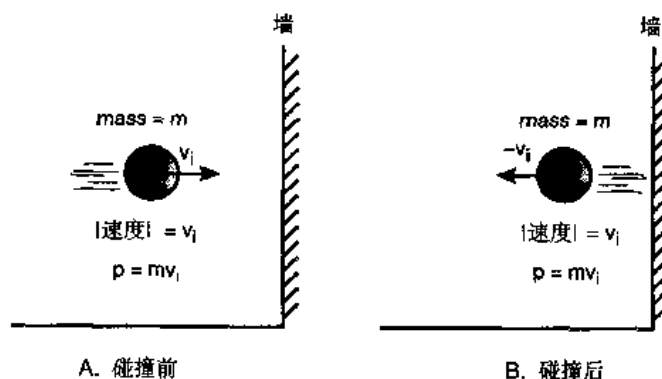


图 13.10 球和墙壁的完全弹性碰撞

然而，我并不对这不完美的世界感兴趣。由于我们是虚拟世界的神，我们可以把事情简单化。因此，现在我要谈谈一维空间中的弹性碰撞，然后再谈谈二维空间的弹性碰撞，其中采用了中世纪的数学理论。我们现在开始。

图 13.11 中有两个方块，A、B 质量分别为 m_a 、 m_b ，速率分别为 v_{ai} 、 v_{bi} 。假定没有摩擦（以后我们会讲到），碰撞为弹性碰撞，那么它们碰撞后会发生什么呢？好，让我们用动量守恒开始。动量守恒表示碰撞前的总动量等于碰撞后总动量。或用数学公式表示：

等式 1：动量守恒

$$m_a v_{ai} + m_b v_{bi} = m_a v_{af} + m_b v_{bf}$$

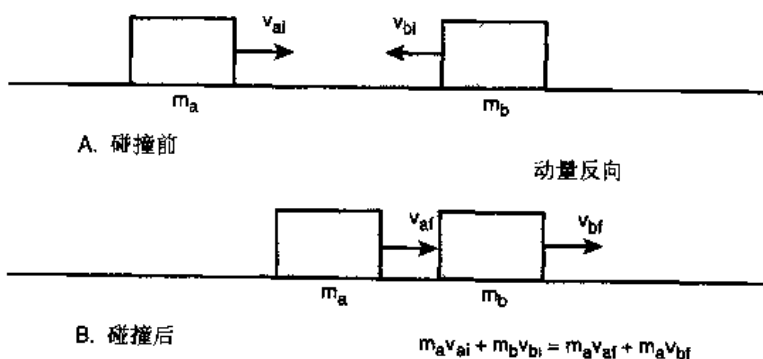


图 13.11 一维空间中两个物体的碰撞反应

好的，已知 m_a 、 m_b 、 v_{ai} 、 v_{bi} ，但我们想要得到最终速率 v_{af} 、 v_{bf} 。问题是现在只有一个

等式，却有两个未知数。这显然是一件很坏的事情。如果知道一个物体的速率，我们就可以把另外一个的速率算出来。但，有没有一种方法不用任何进一步的信息就可以把两个速率都算出来？答案是肯定的！我们可以用另一个物理性质写出一个等式。这个性质就是动能守恒。

动能与动量相似，但与方向无关。它比较像是测量一个系统中的总能量的种类数量。能量是做功的能力，这我会讲到的，在此我们接触不多。计算动能很琐碎，公式为：

公式 2：动能

$$ke = \frac{1}{2}mv^2$$

动量只是 mv ，所以你应该看出动能和动量十分相似，但动能总是正的，用 $\text{kg} \cdot \text{m}^2/\text{s}^2$ 度量。 $\text{kg} \cdot \text{m}^2/\text{s}^2$ 在米-千克-秒系统中我们称之为焦耳 (J)。在任何体系中，碰撞前的动能和碰撞后的动能是一样的，无论碰撞是否是弹性的。当然，为了说明总能量，你要计算出由于变形，热量等原因而损失的能量，但是，如果假定是完全弹性碰撞，碰撞前后的动量只通过物体的速率就可以计算出来。

公式 3：碰撞总能量

$$\frac{1}{2}m_a v_{ai} + \frac{1}{2}m_b v_{bi}^2 = \frac{1}{2}m_a v_{af}^2 + \frac{1}{2}m_b v_{bf}^2$$

与公式 1 组合：

$$m_a v_{ai} + m_b v_{bi} = m_a v_{af} + m_b v_{bf}$$

$$\frac{1}{2}m_a v_{ai} + \frac{1}{2}m_b v_{bi}^2 = \frac{1}{2}m_a v_{af}^2 + \frac{1}{2}m_b v_{bf}^2$$

在这里有两个等式和两个未知数， v_{af} 和 v_{bf} 都可以计算出来。然而，计算相当复杂，所以我只给出结果：

公式 4：每个球的最终速率

$$v_{af} = (2m_b v_{bi} + v_{ai}(m_a - m_b)) / (m_a + m_b)$$

$$v_{bf} = (2m_a v_{ai} - v_{bi}(m_a - m_b)) / (m_a + m_b)$$

最后，回到图 13.11，你可以计算出方块碰撞后的最终速率：

$$m_a = 2 \text{ kg}$$

$$m_b = 3 \text{ kg}$$

$$v_{ai} = 4 \text{ m/s}$$

$$v_{bi} = -2 \text{ m/s}$$

因此，

$$\begin{aligned} v_{af} &= (2m_b v_{bi} + v_{ai}(m_a - m_b)) / (m_a + m_b) \\ &= (2 \times 3 \times (-2) + 4 \times (2 - 3)) / (2 + 3) \\ &= 1.6 \text{ m/s} \end{aligned}$$

$$v_{bf} = (2m_a v_{ai} - v_{bi}(m_a - m_b)) / (m_a + m_b)$$

$$\begin{aligned}
 &= (2 \times 2 \times 4 - (-2) \times (2-3)) / (2+3) \\
 &= 2.4 \text{ m/s}
 \end{aligned}$$

有趣的是，由于物体 A 把大量动量传递给物体 B，两个物体碰撞后都向 X 轴方向运动，如图 13.11.B 所示。

刚才所做的显示了怎样利用动量和动能来解决动力学问题。然而，在二、三维空间中这个问题更加复杂。这样的碰撞研究称为碰撞反应，这会在这一章的后面部分讲到，后面部分还会讲到在 2D 空间中的完全碰撞和非完全碰撞。可是现在，只关注动量。

万有引力效果模型

万有引力效果是游戏程序设计员在游戏中需要建立的最普通的效果之一。万有引力是宇宙中任何一个物体吸引其他物体的力。万有引力是看不见的力，也不像磁场那样可以被中断。

实际上，万有引力不是真正的力。这只是我们对它的感觉。万有引力是由于空间曲率引起的。当任意一个物体放在空间里，它会引起周围空间的弯曲，如图 13.12 所示。周围空间的弯曲会引起势能的不同，因此在自流井旁边的任何物体都会向这个物体“降落”。奇怪吧？这就是万有引力。万有引力是时空结构弯曲的表现。

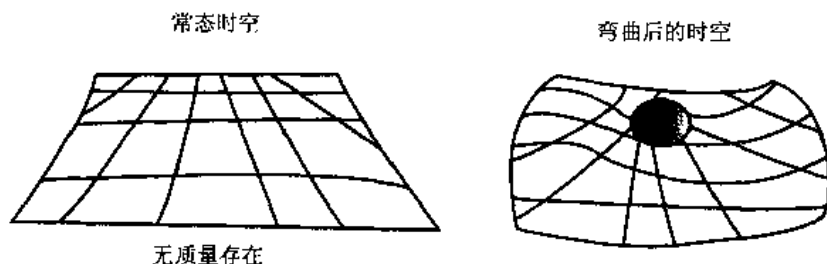


图 13.12 万有引力和时空

你不用过多地考虑时空弯曲和万有引力真正是什么；你只是想建立一个万有引力模型。当你建模时，有两种情况需要考虑：

- 情况 1：两个或两个以上的物体有相近质量。
- 情况 2：两个物体，其中一个物体的质量比另一个物体的质量大很多。

情况 2 是情况 1 中的特例。例如，在学校里你可能学过：如果你把一个篮球和一个电冰箱从楼上扔下，篮球和电冰箱下降的速度是一样的。事实却不是这样的，只是由于篮球和电冰箱速度相差极小（大约 10^{-24} ），你根本就看不出它们的差别。当然，还有其他因素使得它们的速度不一样，如风切力和摩擦力，因此一个篮球比一张纸下降速度快得多，因为纸受到风阻力比较明显。

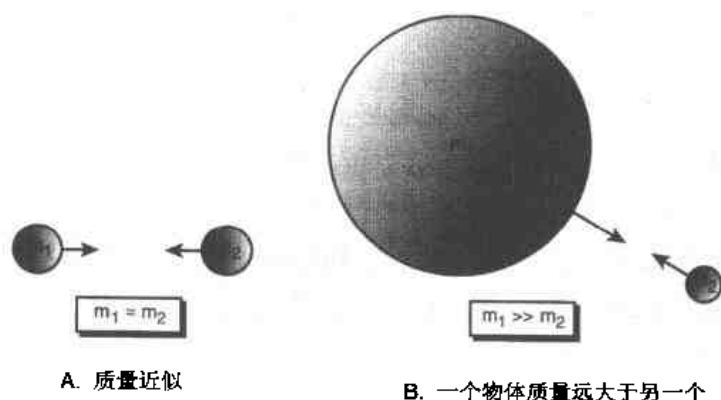


图 13.13 两个普通的万有引力例子

现在你对万有引力有了一个基本的了解，让我们来看看关于万有引力的数学公式。任何两个物体之间（质量分别为 m_1 、 m_2 ）的万有引力为：

$$F = Gm_1m_2/r^2$$

这里 G 为宇宙里的万有引力常数，等于 $6.67 \times 10^{-11} \text{N} \cdot \text{m}^2 \cdot \text{kg}^{-2}$ 。同样质量用千克，距离 r 用米。现在来算算两个 70 公斤人在距离 1 米时，他们之间的万有引力：

$$F = 6.67 \times 10^{-11} \text{N} \times 70 \text{kg} \times 70 \text{kg} / (1 \text{m})^2 = 3.26 \times 10^{-7} \text{N}$$

是不是很小？然而，我们用一个人和一个行星地球来同样的实验，距离也为 1 米，地球质量为 $5.98 \times 10^{24} \text{kg}$ ：

$$F = 6.67 \times 10^{-11} \text{N} \times 70 \text{kg} \times 5.98 \times 10^{24} \text{kg} / (1 \text{m})^2 = 2.79 \times 10^{16} \text{N}$$

显然， 10^{16}N 的力可以把你摔成肉饼，所以肯定是你在哪里算错了。问题是你把地球看作一个点，只有 1 米的距离。最好的近似值采用地球的半径作为距离，距离为 $6.38 \times 10^6 \text{m}$ ：

数 学

你可以假设任意一个半径为 r 的球形物质为一个质点，条件是组成球的物质是同质的，在计算时必须把另外一个物体放在大于或等于 r 的位置上。

$$F = 6.67 \times 10^{-11} \text{N} \times 70 \text{kg} \times 5.98 \times 10^{24} \text{kg} / (6.38 \times 10^6 \text{m})^2 = 685.93 \text{N}$$

现在看起来更为合理。在地球上 1 磅等于 4.45N，把力用磅表示出来：

$$685.93 \text{N} / (4.45 \text{N} / 1 \text{lb.}) = 155 \text{lbs}$$

这刚好等于最初的重量！无论如何，现在你已经知道怎样计算两个物体之间的力，你可以把这个简单的模型运用到游戏当中去。当然，你不必使用真正的引力常数 $G = 6.67 \times 10^{-11}$ ，你可以随意设定，你就是上帝。惟一重要的是等式的形式，等式显示了两个物体之间的万有引力是成比例的——为两个物体质量的乘积除以物体中心之间距离的平方。

重量井模型

使用了前面部分解释过的公式后，你可能想在空间游戏中模拟一个黑洞。例如，你有一艘船在一个黑洞附近行驶，你想使得这艘船一靠近黑洞就会沉没。这时可使用万有引力公式，你要设定能够很好运用在虚拟游戏世界中的常量 G （基于屏幕分辨率、帧频等），然后任意设置船的质量和黑洞的质量。黑洞的质量应该比船的质量大得多。然后你需要计算力，并通过力用公式 $F=ma$ 把加速度算出来。你可以简单的引导或驾驶船向黑洞行驶。随着船的靠近，力也加大，直到船沉没。

DEMO13_1.CPPICPP 中有一个黑洞模拟（有两个物体，一个比另一个大许多）。这是个空间模拟程序，你可以在屏幕上操纵一艘船，但在屏幕中间有一个黑洞，你必须要注意到。用箭头键控制船。看看你掉进黑洞后会是怎样！

在游戏中使用万有引力的下一个地方，就是使一个物体从空中或楼上以适当的速率落下。这就是我们以前说过的一个例子的特例，即一个物体比另一个物体的质量大得多。然而，这还有一个约束条件——其中一个物体是固定的，即地面。我所描述的情况如图 13.14 所示。

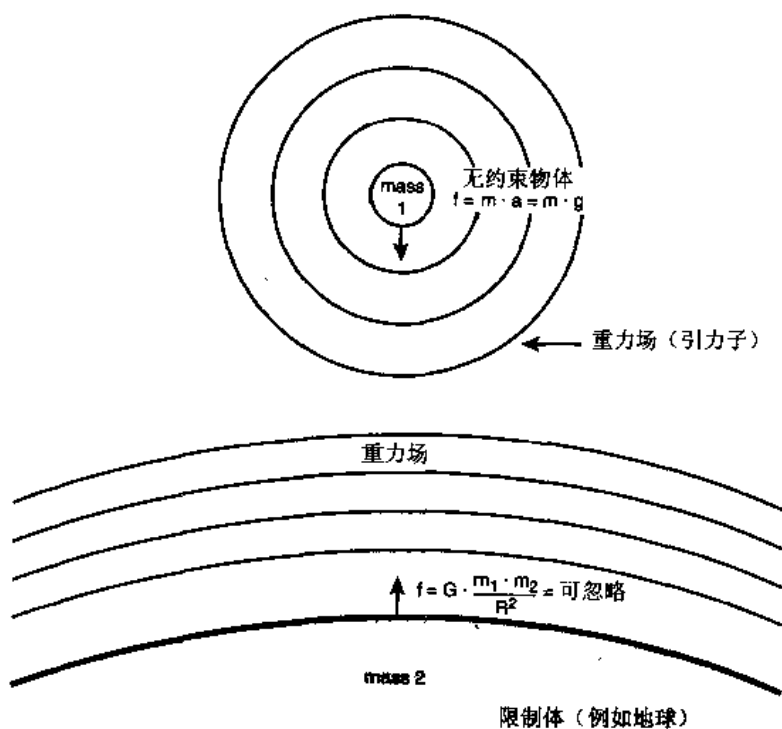


图 13.14 万有引力

在这个例子中，我们可以假设几个条件，这样会使计算简单一些。第一个条件就是落下的物体的加速度（由于重力引起的）是恒定的，并等于 9.8m/s^2 或 32ft/s^2 。当然，这不是

真实值，但也有 23 个小数位。如果你知道任何物体的加速度仅仅等于 9.8m/s^2 ，那么代入我们的运动位置或速率公式中。因此，速率作为含重力的时间函数是：

$$v(t) = v_0 + 9.8\text{m/s}^2 \cdot t$$

位置是：

$$y(t) = y_0 + v_0 \cdot t + \frac{1}{2} \cdot 9.8\text{m/s}^2 \cdot t^2$$

在球从楼上落下的例子中，我们可以设初始位置 x_0 等于 0，初始速率也等于 0。这样可以简化下降物体模型为：

$$y(t) = \frac{1}{2} \cdot 9.8\text{m/s}^2 \cdot t^2$$

而且，你可以用任意数值代替常数 9.8， t 代表游戏中的帧数。把这些全部考虑进去。通过以下代码，就可以使一个物体从屏幕上端落下：

```
int y_pos = 0, // top of screen
y_velocity = 0, // initial y velocity
gravity = 1, // do want to fall too fast
// do gravity loop until object hits
// bottom of screen at SCREEN_BOTTOM
while (y_pos < SCREEN_BOTTOM)
{
    // update position
    y_pos += y_velocity;
    // update velocity
    y_velocity += gravity;
} // end while
```

技巧

我用速率来修改位置而不是直接用位置公式来修改。因为这更简单。

你可能要问怎样使物体曲线降落。这很简单——只要使物体沿 x 方向以恒定的速度运动，物体就会看起来像是抛出去的，而不只是下降。以下的代码可以使物体曲线下降。

```
int y_pos = 0, // top of screen
y_velocity = 0, // initial y velocity
x_velocity = 2, // constant x velocity
gravity = 1, // do want to fall too fast
// do gravity loop until object hits
// bottom of screen at SCREEN_BOTTOM
while (y_pos < SCREEN_BOTTOM)
{
    // update position
    x_pos += x_velocity;
    y_pos += y_velocity;
    // update velocity
```

```
y_velocity+=gravity;
} // end while
```

导弹轨道模型

我们已经可以使物体垂直降落，现在我们来对视频游戏程序做一些适当的改变。计算导弹轨道路径好吗？图 13.15 显示了对这个问题的通常调整。有个地平面，设置 $y=0$ ，一个坦克定位在 $x=0$ ， $y=0$ ，炮筒的与 X 方向的角度为 θ 。问题是，如果我们发射一个质量为 m 速率为 v_i 的导弹，会发生什么情况呢？

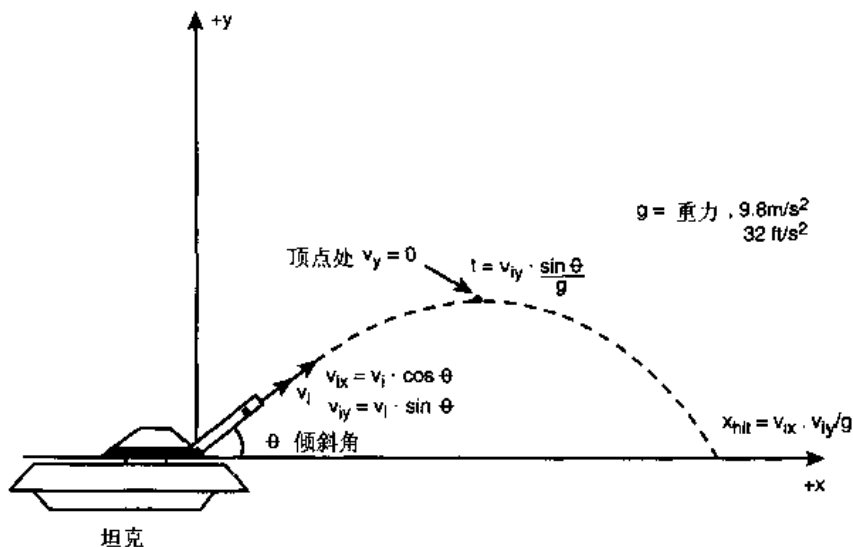


图 13.15 轨迹问题

我们可以通过把导弹的速率分为 x 、 y 两个分量来解决问题。首先，我们把速率分解为 (x, y) 矢量：

$$v_{ix} = v \cdot \cos \theta$$

$$v_{iy} = v \cdot \sin \theta$$

好的，现在把 x 分量放一边，来想想这个问题。导弹将会上升，然后下降，最后击中地面。这些要发多少时间呢？看看以前说过的重力公式：

$$v(t) = v_0 + 9.8 \text{ m/s}^2 \cdot t$$

y 轴的位置：

$$y(t) = y_0 + v_0 t + 1/2 \cdot 9.8 \text{ m/s}^2 \cdot t^2$$

第一个公式告诉我们速率和时间相关。这就是我们所需要的。我们知道当炮弹达到最高点时，它的速率等于 0。而且，炮弹达到最高点所需要的时间等于炮弹从最高点下降到地面所用的时间。看看图 13.15。代入我们所设定的炮弹初始 y 速率，解出 t ：

$$v_y(t) = v_{iy} - 9.8 \text{ m/s}^2 \cdot t$$

注意到由于是重力，我转换了加速度的符号，因为向下的方向为负，通常，当速率等于0时：

$$0 = v \cdot \sin \theta - at \quad (a \text{ 只是加速度})$$

解出 时间 t ，我们得到：

$$t = v_{iy} \cdot (\sin \theta) / a$$

由于导弹的飞行肯定是上升然后下降，所以飞行的总时间为上升时间加上下降时间等于 $t+t=2t$ 。为此，我们现在回过头来看看 x 分量。我们知道飞行总时间为 $2t$ ，我们可以通过 $(v_{iy} (\sin \theta) / a)$ 计算出 t 。因此，导弹在 x 轴方向飞行的距离是：

$$X(t) = v_{ix}t$$

代入数值，结果为：

$$x_{hit} = (v \cdot \cos \theta) \cdot (v \cdot (\sin \theta) / a)$$

或

$$x_{hit} = v_{ix} \cdot v_{iy} / a$$

数 学

注意到我用 a 替换 9.8 作加速度。我这样做是由于加速度只是一个数值，你可以设置为任意值。

这些只是物理知识，但怎样在程序里建模呢？你所需要做的就是给炮弹设置一个 x 轴方向的恒定速率， y 轴方向的重力及测试出什么时候炮弹击中地面或其他地方。当然，实际中，空气阻力会降低 X 和 Y 方向的速率，但在算法中可以忽略这些。以下就是这个例子的代码：

```
//Input
float x_pos    =0 , // starting point of projectile
    y_pos     =SCREEN_BOTTOM, //bottom of screen
    y_velocity =0, // initial y velocity
    x_velocity =0, // constant x velocity
    gravity    =1, // do want to fall too fast
    velocity   = INITIAL_VEL, // whatever
    angle      = INITIAL_ANGLE; // whatever, must be in radians
// compute velocities in x,y
x_velocity =velocity*cos(angle);
y_velocity =velocity*sin(angle);
//do projectile loop until object hits
// bottom of screen at SCREEN_BOTTOM
while(y_pos<SCREEN_BOTTOM)
{
    // update position
    x_pos+=x_velocity;
    y_pos+=y_velocity;
    // update velocity
```

```
y_velocity+=gravity;
} // end while
```

这就是所有的建模代码！如果你想加点风力，只是在 x 轴运动方向上附加一个很小的加速度，假设风力能够对炮弹产生一个与 x 轴运动方向相反的恒定加速度。因此，你只要在炮弹飞行中加入下面的代码就可以了：

```
x_velocity-=wind_factor;
```

这里 wind_factor 可以近似等于 0.01——相当小。

CD 中的 DEMO13_4.CPPIEXE 是所有炮弹轨迹的演示，如图 13.16 所示。演示中你可以用虚拟的炮筒瞄准，发射炮弹。

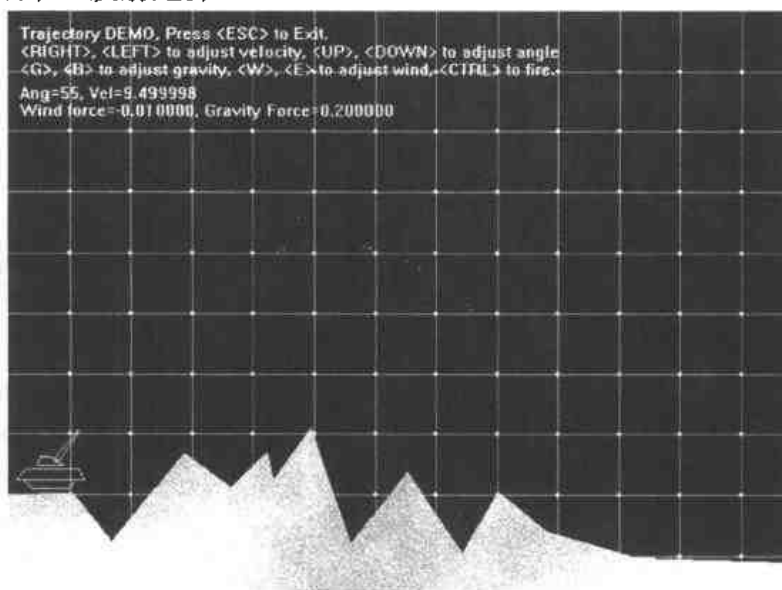


图 13.16 发射炮弹演示

下面是控制键：

键	作用
上，下键	控制坦克炮筒的角度
左，右键	控制炮弹的速率
G，B 键	控制重力
W，E 键	控制风速
Ctrl 键	发射炮弹

摩擦力

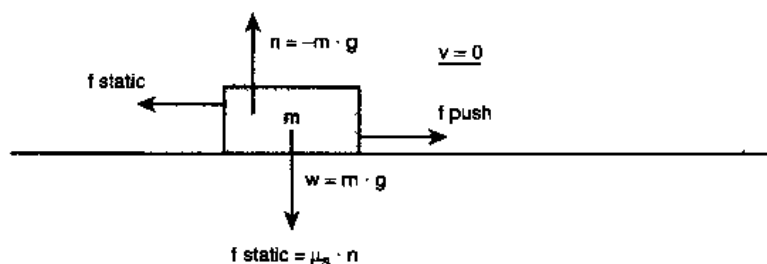
下一个讨论的主题是摩擦力，摩擦力是另一个体系中任何阻碍和消耗能量的力。例如，

汽车使用内燃工作，然而，由于热转换或机械摩擦，损失了 30%~40% 的能量。另一方面，自行车的效率为 80%~90%，可能是现存效率最高的交通工具。

摩擦基本概念

基本上来说摩擦是与运动相反方向的阻力，因此可以用一个力（通常是摩擦力）来模拟。图 13.17 描述了一个质量为 m 的物体在平面上的标准摩擦模型。

A. 静止状态，无运动



B. 运动状态，物体在移动

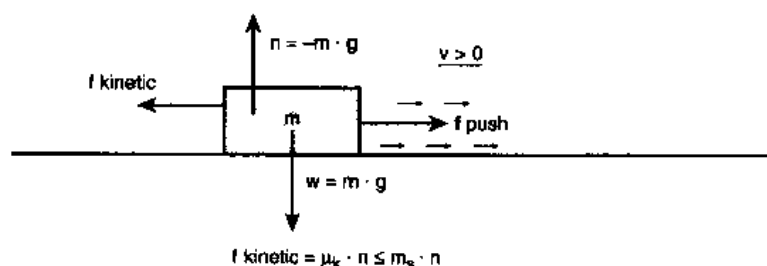


图 13.17 基本的摩擦模型

如果你以平行于平面的方向推动物体，你将会遇见一个与推动方向相反的阻力或摩擦力。这个力的数学公式为：

$$F_{\text{fstatic}} = mg\mu_s$$

其中， m 为物体的质量， g 为引力常数 (9.8m/s^2)， μ_s 为静态摩擦系数。 μ_s 与物体和平面的材料和状态有关。如果你对物体施加的力 F 大于 F_f ，那么物体就会开始运动。一旦物体处于运动状态，它的摩擦系数通常会减小为另外一个常数，这个常数我们称之为动摩擦系数 μ_k 。

$$F_{\text{kinetic}} = mg\mu_k$$

如果你不再对物体施加力，由于摩擦力一直存在，物体会慢慢减速，直到停下。

在平面上建立一个摩擦力模型，你只要给物体加上一个恒定负方向的速率，这个速率与你想要的摩擦力成比例。

数学公式为：

$\text{Velocity New} = \text{Velocity Old} - \text{friction}$

所以一旦你停止移动物体，物体就会以一个恒定的数值减速。当然，你必须注意要把速率的符号设置为反方向或其他方向，但这只是一个小细节。下面有个例子，例子中有个物体向右移动，初速率为 16 像素每帧，由于虚拟摩擦力，物体以 1 像素每帧减速：

```
int x_pos    -0,  // starting position
    x_velocity =16, // starting velocity
    friciton  =-1, // fricitonal value
// move object until velocity <=0
while(x_velocity>0)
{
    // move object
    x_pos+=x_velocity

    // apply friction
    x_velocity+=fricition;
} // end while
```

首先，你应该注意到摩擦力模型和万有引力模型很相似。它们几乎就是一样的。重力和摩擦力都是以同样的方式作用于物体。总之，宇宙中所有的力都可以以完全一样的方式建模。同样，你可以对物体施加任意多个摩擦力。最后把这些摩擦力加起来。

我编写了一个小空气曲棍球的演示作为使用摩擦力的例子，名为 DEM013_5.CPPIEXE。如图 13.18 所示。演示中，每次当你按下空格键时，就会在虚拟空气曲棍球桌面上发射出一个曲棍球圆盘，这个曲棍球圆盘的方向是随机的。如果你想改变桌面的摩擦系数，可以使用方向键。

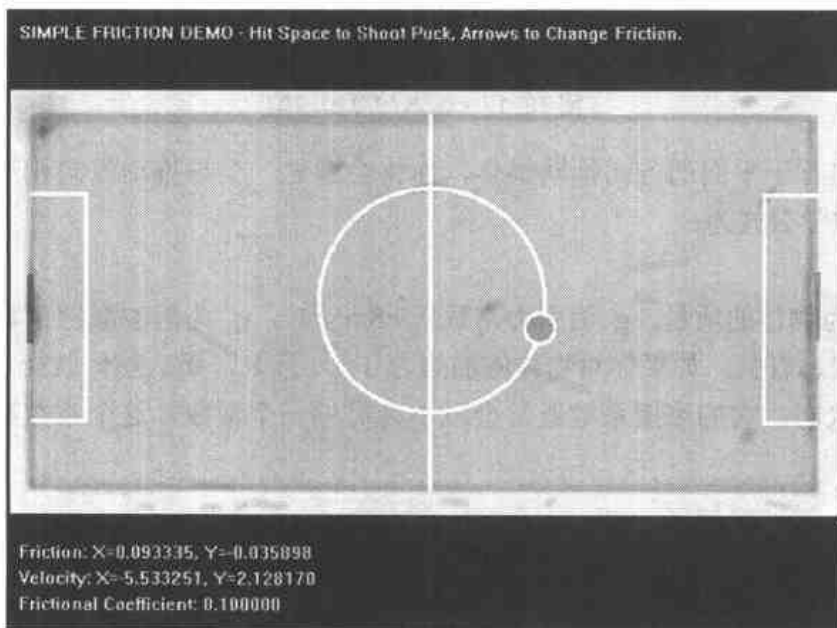


图 13.18 曲棍球演示

斜面上的摩擦力

摩擦力可以作为一个简单的阻力或物体的负速率来建模。然而，我想给你看看斜面上的数学和摩擦力的演算，因为这可以使你以后能够分析更为复杂的问题。注意：我将使用多个矢量，所以如果你还是对此不熟悉那就看看我在第 8 章“矢量光栅化及 2D 变换”中介绍的矢量问题或随便看一本好的线性代数书。

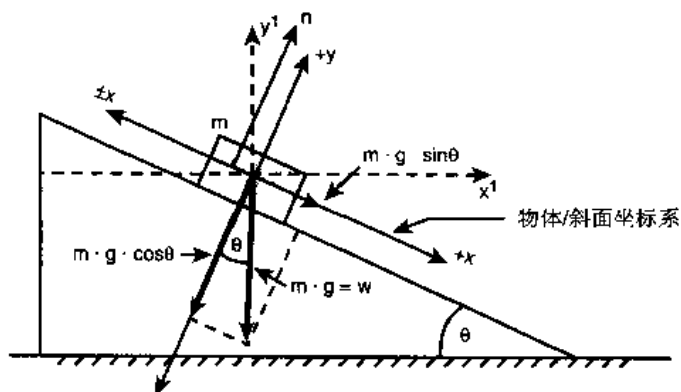


图 13.19 斜面问题

图 13.19 描述了我们解决的问题。基本上，就是一个质量为 m 的物体放在一个斜面上。斜面的静态摩擦系数和动态摩擦系数分别为 μ_s 和 μ_k 。我们需要做的第一件事就是写出描述物体处于平衡位置（也就是静止状态）的公式。本例中， x 轴方向力的总和等于零， y 轴方向力的总和也为零。

在推导之前我们必须先接触一个新的概念——法向力。法向力就是斜面对物体的推回力，换句话说，如果你的体重为 200 磅，那么就有一个 -200 磅的法向力把你推回（由于你所站位置的表面张力）。我们通常用 n 表示法向力。法向力在数量上等于：

$$n = mg$$

很有趣是吧？但在坐标系中，法向力肯定和重力是相反的，或

$$n - mg = 0$$

这就是为什么所有的物体没有陷入到地下的原因。好的，既然我们已经了解了法向力，那我们开始推导物体运动公式。首先，我们在斜面上放一个坐标系，坐标系的 $+x$ 轴与斜面平行，沿向下滑动的方向。坐标系有助于数学计算。那么我们写出 x 轴和 y 轴的平衡公式。在 x 轴上，我们知道作用于物体的重力分量是：

$$\text{重力} = mg \sin \theta$$

防止物体下滑的摩擦力为：

$$\text{摩擦力} = -n \mu_s$$

负号表示力作用于相反的方向。如果物体不滑动，所有力的总和为 0。数学公式为：

重力+摩擦力=0

或 x 轴方向力的总和为:

$$\Sigma F_x = mg \sin \theta - \eta \mu_s = 0$$

用同样的方法推导出 y 轴上的公式, 这相当简单, 因为在 y 轴上只有重力和法向力:

$$\Sigma F_y = \eta - mg \cos \theta = 0$$

数 学

注意到我用正弦和余弦来分解力在 x 轴和 y 轴上的分量。我只是把力矢量分解成几个分量, 没有其他原因。

好了, 把公式组合起来, 我们得到:

$$\Sigma F_x = mg \sin \theta - \eta \mu_s = 0$$

$$\Sigma F_y = \eta - mg \cos \theta = 0$$

但 η 等于多少? 从 ΣF_y 中, 我们得到:

$$\eta - mg \cos \theta = 0$$

因此,

$$\eta = mg \cos \theta$$

因此, 我们可以写出:

$$\Sigma F_x = mg \sin \theta - mg \cos \theta \mu_s = 0$$

这就是我们想要得到的。从这个公式我们可以得到以下结果:

$$mg \sin \theta = mg \cos \theta \mu_s$$

$$\mu_s = (mg \sin \theta) / (mg \cos \theta) = \tan \theta$$

得到:

$$\theta_{\text{critical}} = \tan^{-1} \mu_s$$

注意, 上式给出了一个临界角 (θ_{critical})。如果斜面的倾斜角度达到临界角, 物体将会滑动。临界角等于静态摩擦系数的反正切。如果我们不知道物体和斜面的摩擦系数, 那么我们可以使平面倾斜, 直到物体开始滑动, 得到开始滑动时平面的倾斜角度。然后通过计算得到摩擦系数。但这个公式对 x 轴分析没有什么帮助。这个公式告诉我们当斜面角度小于 θ_{critical} 时, 物体是不会运动的。当斜面角度达到 θ_{critical} 时, 物体开始滑动, 而且滑动是由下面这个公式控制的:

$$\Sigma F_x = mg \sin \theta - \eta \mu_s$$

当物体开始滑动时, 不同的是 $mg \sin \theta - \eta \mu_s > 0$, 但我们还要把摩擦系数该为 μ_k (动态摩擦系数)。

$$\Sigma F_x = mg \sin \theta - \eta \mu_k$$

技巧



你可以取 μ_s 和 μ_k 的平均值, 在所有的计算中都用这个平均值。因为你只是做视频游戏, 并不是真正的模拟, 因此把两个系数简化为一个, 并没有什么影响。但如果需要精确一点, 你应该在适当的时候使用两个摩擦系数。

把上面的公式记住, 我们来计算出 x 轴方向的最终力。我们知道 $F=ma$, 因此:

$$F_x = ma = mg \sin \theta - (mg \cos \theta) \mu_k$$

消掉 m , 得到:

$$a = g \sin \theta - g \cos \theta \mu_k$$

$$a = g (\sin \theta - \cos \theta \mu_k)$$

你可以使用这个准确的模型来移动物体, 也就是, 每一次, 你可以通过 $g (\sin \theta - \cos \theta \mu_k)$ 增加物体在 x 轴正方向的速率。这有一个问题: 这个问题能用我们的循环坐标系解决! 有个窍门可以避免这个问题——知道斜面的角度, 因此可以算出沿斜面下降角度的矢量:

$$x_{\text{plane}} = \cos \theta$$

$$y_{\text{plane}} = -\sin \theta$$

$$\text{Slide_Vector} = (\cos \theta, -\sin \theta)$$

这个负号在 y 分量上, 因为它处于 $-y$ 方向。用这个矢量我们每次都可以使物体向正确的方向移动——这是一个辅助手段, 但也可以用。下面的代码可以执行平移和速率跟踪。

```
// Inputs
float x_pos      =SX, // starting point of mass on plane
y_pos          =SY,
y_velocity      =0, // initial y velocity
x_velocity      =0, // initial x velocity
x_plane         =0, // sliding vector
y_plane         =0,
gravity         = 1, // do want be greater
velocity        = INITIAL_VEL, // whatever
// must be in radians and it must be greater
// than the critical angle
angle           =PLANE_ANGLE, //compute velocities in x, y
frictionk       =0.1; // frictional value
// compute trajectory vector
x_plane =cos(angle);
y_plane =sin(angle); // no negative since +y is down
// do silde loop untile object hits
// bottom of screen at SCREEN_BOTTOM
while(y_pos<SCREEN_BOTTOM)
{
    // update position
    x_pos+=x_velocity;
    y_pos+=y_velocity;
    // update velocity
    x_vel+=x_plane*gravity*(sin(angle) -frictionk *cos(angle));
    y_vel+=y_plane*gravity*(sin(angle) - frictionk *cos(angle));
} // end while
```

物理模型的关键有时就是知道一些基本物理知识，这样才能使你用让人信服的方式建模。在斜面的例子中，基本上把所有的数学简化成一个概念——加速度是角度的函数（我们通过常识就能知道）。然而，在第二册中，我将用数值积分介绍更多的实际物理学知识，在那些例子中，你需要知道一切事物的实际模型和作用在物体上的实际力。

基本的特殊碰撞反应

前面，我介绍过有两种碰撞：弹性碰撞和非弹性碰撞。弹性碰撞中碰撞物体的动能和动量都守恒。而非弹性碰撞中，碰撞物体的动能和动量不守恒，有些能量转化为热量或用作机械变形。

大多数视频游戏中根本不采用非弹性碰撞，而坚持采用弹性碰撞，因为非弹性碰撞的计算非常麻烦。在我告诉你怎样作之前多开动一下脑筋。对弹性碰撞和线性弹性碰撞一窍不通的游戏程序师能够仿造出碰撞，我们也可以。

简单的反弹

图 13.20 描述了游戏中的一个相当普遍的碰撞问题，即把球从屏幕的边界反弹回来。假设物体的初速率为 (sv, yv) ，物体能够击中屏幕的 4 面。如果物体与一个质量比它大得多的另一个物体相撞，那么问题就会简单得多，因为我们只需要计算出一个物体碰撞后的情况，而不是两个。撞球台就是一个很好的例子，球的质量相对于撞球台来说是非常小的。

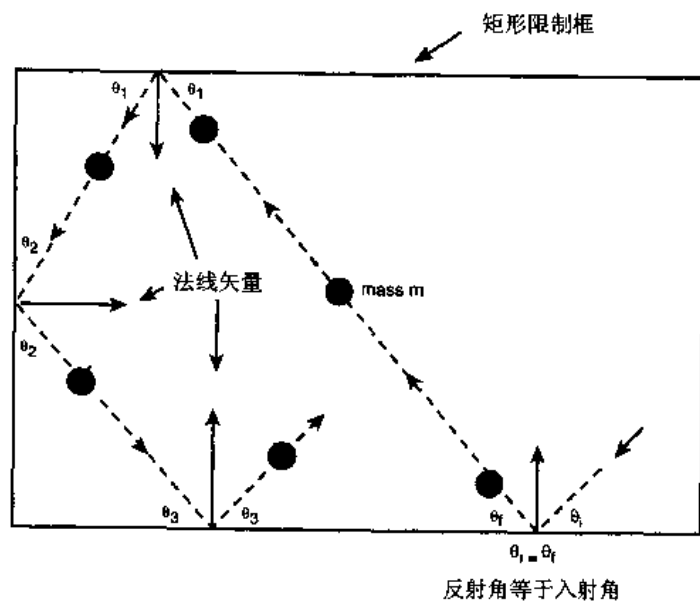


图 13.20 反弹的球

当球撞到球台一边时，球就会从球台的这边弹回，弹回的角度等于开始运动轨迹的角度，并与之相反。如图 13.20 所示。因此，我们需要将一个物体从撞球台状的环境中弹回（这个环境由质量很大的硬边组成），并计算出物体运动一般矢量方向。这个方向包括物体撞击时和弹回后的方向。如图 13.21 所示。撞击时的角度与弹回后的角度相等。

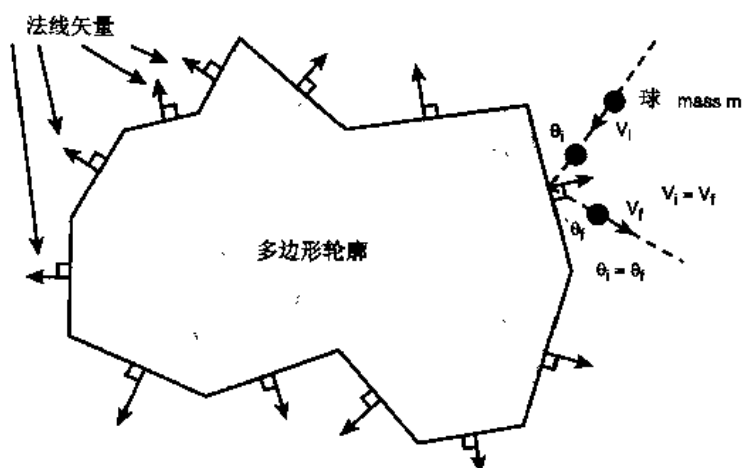


图 13.21 球在不规则物体上的反弹

虽然上面所说的没有一般的弹性碰撞那么复杂，但还是需要一些三角学知识，这样才会更简单。当然使问题简单化的窍门就是把你所需要建立物理模型了解透彻。既然你了解了所有的条件的相关知识。那么你可以看看你是否能够通过其他方式解决这个问题。下面是解决问题的技巧：对问题的思考不依据角度而是依据结果。如果物体撞在边界的正东面或正西面，那么你可把 x 速率反向，而不用考虑 y 速率。同样在北面和南面中，你可把 y 速率反向，而不用考虑 x 速率。下面是代码：

```
// given the object is at x,y with a velocity if xv,yv
// test for east and west wall collisions
if(x>EAST_EDGE || x<WEST_EDGE)
    xv=-xv; // reverse x velocity
// now test for north and south wall collisions
if (y>SOUTH_EDGE || y<NORTH_EDGE)
    yv=-yv; // reverse y velocity
```

当然，这只能简化垂直和水平方向上的反弹。你将必须用更为普通的角度计算与 x 轴和 y 轴不成联合线性关系的边界。

技巧

如果你想采用以前的技术仅仅使物体互相弹回，那么假定每个物体从其他物体角度来看都是一个边界长方形。使物体发生碰撞，并重新计算速率。如图 13.22 所示。

我编写了一个演示，名为 DEMO13_6.CPPIEXE，这个演示是作为这些技术的示范。演示中有个撞球台模型，上面有一些不停跳动的球。图 13.23 显示了一个游戏。注意到游戏中的球与球并不相撞，只是球和球台边相撞。

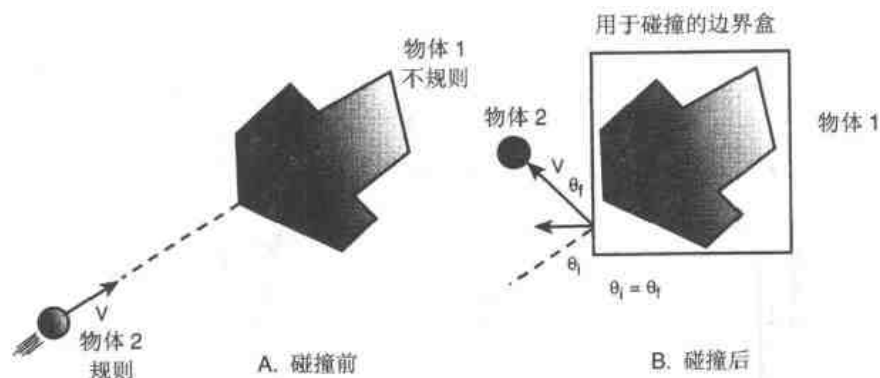


图 13.22 物体之间的碰撞的简化

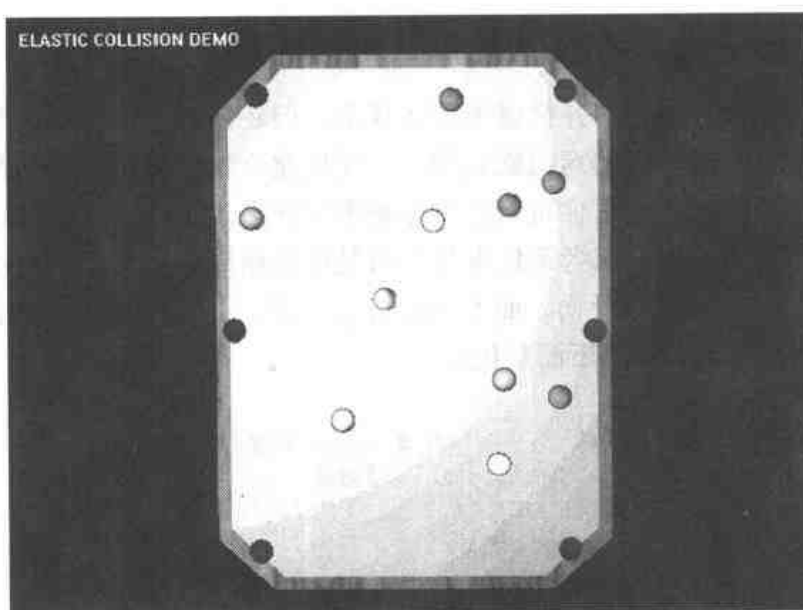


图 13.23 简单的碰撞球台模型

计算任意方向上的平面碰撞反应

如果你想写一个比较差的游戏，使用长方形作为边界碰撞容器就可以了，但现在都 21 世纪了，我们要求应该更高一些！我们需要做的就是推导出一个适用于平面上矢量反射的反射公式，如图 13.24.A 所示。图 13.24.A 处于 2D 环境（3D 环境也是一样的）。解决这个

问题, 首先我们必须做一个假设, 这个假设是当一个很小物体完全弹性碰撞到边界会发生什么情况呢? 我想我们已经可以得出结论: 物体弹回的角度和碰撞前的角度相同。因此, 相对于垂直于边界的反射角度 (物体碰撞后离开边界的角度) 和入射角度 (碰撞前的入射角度) 相等。现在, 我们来看看相应的数学知识。

解决这个问题只需要一个向量几何结构图, 但这并不很琐碎。

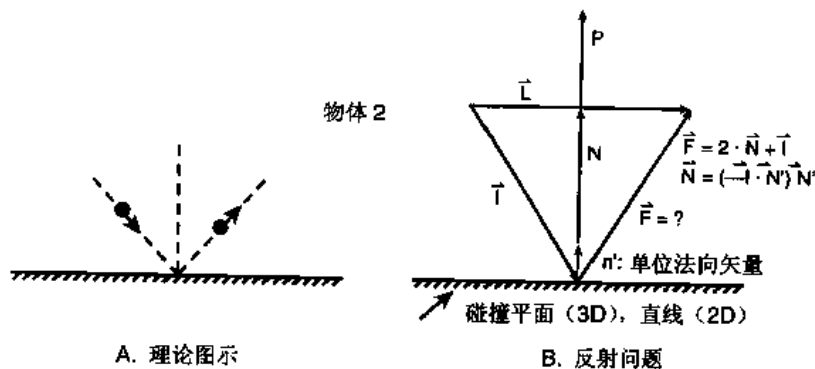


图 13.24 矢量反射问题

提示

如果你想得到一份游戏编程的工作, 我敢肯定公司在面试的时候会问你这个问题, 因为这个问题颇具迷惑性。幸运的是你刚好看了这部分, 能够对这个问题脱口而出, 公司会认为你是个天才。我们现在开始吧。

图 13.24.B 描述了这个问题的。注意到在这没有 x 轴和 y 轴。由于我们使用矢量, 所以就不需要 x 轴和 y 轴, 同时我也想使问题更为一般化。

这个问题可以这样描述:

给定一个初始矢量方向 \vec{I} 及垂直平面的法线 \vec{N}' 。确定 \vec{F} 。

现在我们来说说法向矢量。法向矢量 \vec{N}' 只是 \vec{P} 的标准形式。但 \vec{P} 是什么呢? \vec{N} 只是垂直于平面或直线 (就是球击中并弹回的地方) 的垂线。我们可以用一些方式计算出 \vec{P} ; 可以预先计算出 \vec{P} , 然后把它存储到一个数据结构中, 或在空闲时计算。

不同的边界有不同计算 \vec{P} 的方式。如果边界处于 3D 环境中, 那么我们可以根据平面的点法式得到 \vec{P} :

$$n_x(x-x_0) + n_y(y-y_0) + n_z(z-z_0) = 0$$

法向矢量就是 $\vec{P} = \langle n_x, n_y, n_z \rangle$ 。为确定法线是否标准化或是一个单元矢量, 那么就要把总长度划分为几个单元。

$$\vec{N}' = \langle n_x, n_y, n_z \rangle / |\vec{P}|$$

式中的 $|\vec{P}|$ 是长度, 可以通过下面的公式计算出来:

$$|\vec{P}| = \sqrt{n_x^2 + n_y^2 + n_z^2}$$

通常, 矢量的长度是矢量分量的平方和的平方根。

另一方面, 如果边界处于 2D 环境, 你可以通过任意垂直于边界的矢量计算出法线。因此, 如果在 2D 环境中, 线以 2 个端点的形式存在, 如图 13.25 所示:

设定: $P_1(x_1, y_1)$, $P_2(x_2, y_2)$

那么, 从 P_1 到 P_2 的矢量为

$$V_{12} = \langle x_2 - x_1, y_2 - y_1 \rangle = \langle v_x, v_y \rangle$$

用这个技巧可以得到垂线:

$$P_{12} = \langle v_x, v_y \rangle = \langle -v_y, v_x \rangle$$

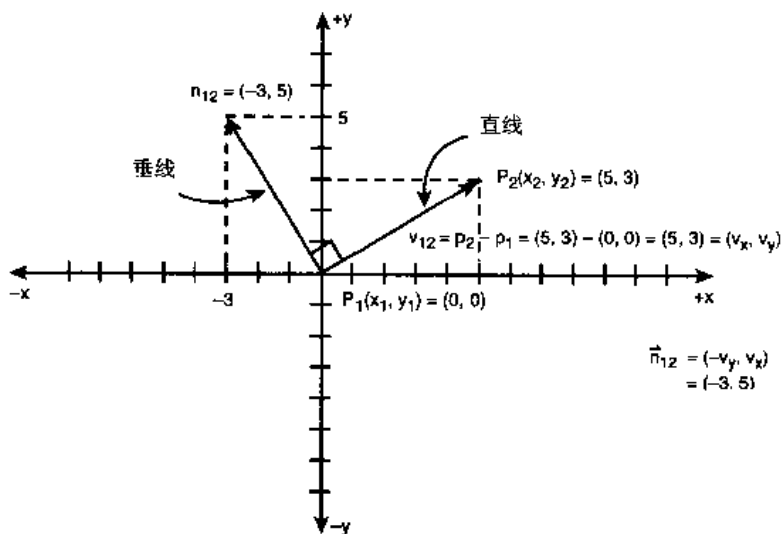


图 13.25 计算一条线的垂线

这个技巧是基于对点积的定义, 点积表示矢量和它的法线的点乘等于 0, 因此:

$$V_{12} \cdot N_{12} = 0$$

$$\langle v_x, v_y \rangle \cdot \langle n_x, n_y \rangle = 0$$

或

$$v_x \times n_x + v_y \times n_y = 0$$

使得上式成立的条件是:

$$n_x = -v_y, n_y = v_x$$

$$v_x \times (-v_y) + v_y \times (v_x) = -v_x \times v_y + v_x \times v_y = 0$$

好的, 你已经知道怎样得到法向矢量, 当然你需要使之标准化, 确定它的长度等于 1, 所以

$$N' = P / |P| = \langle -v_y, v_x \rangle / \sqrt{(-v_y)^2 + v_x^2}$$

在这点上我们得到法向矢量 N' , 由于 N 沿着 N' , 所以在图上不要把 N' 与 N 混淆, 但 N' 与 P 没有任何关系。 N 是 I 沿着 N' 的投影。投影类似阴影。如果我用一束光照射物体的左侧, 光的方向是从左到右, 那么 N 就是 I 的在 N' 轴上的阴影或投影。这个投影就是我们

需要的 N 。

求出 N 后，我们可以用向量几何得到 F ，首先， N 等于：

$$N = (-I \cdot N')N'$$

从公式中可以看出 N 等于 $-I$ 和 N' 的点积，并乘以 N' 。我们把这个公式拆成两部分。第一部分 $(-I \cdot N')$ 只是一个标量长度；不是矢量。它是点积的一个方便的表达形式；如果你想得到一个矢量（垂线）的阴影，那么你可以用这个矢量同它的矢量单位点积，因此你可以得到任意方向上的矢量分量。你可能要问“在 V' 方向的 R 是什么？”“在哪使 V' 标准化？”因此，第一部分 $(-I \cdot N')$ 给你一个值（ -1 只是 I 方向的倒转）。但，你需要一个矢量 N ，所以你要把这个数值与单位矢量相乘 N' （矢量乘法），然后就得到 N 。

N 是所有的的基础，所以你得到 N 后，仅仅通过一些矢量几何计算就可以得到 F ：

$$L = N + I$$

$$F = N + L$$

得出：

$$F = N + (N + I)$$

所以，

$$F = 2N + I$$

牢牢记住最后一个公式。

矢量反射实例

图 13.26 描述了这个问题的设置。使一个反弹平面与 x 轴成线形，这样可以使问题简单化。

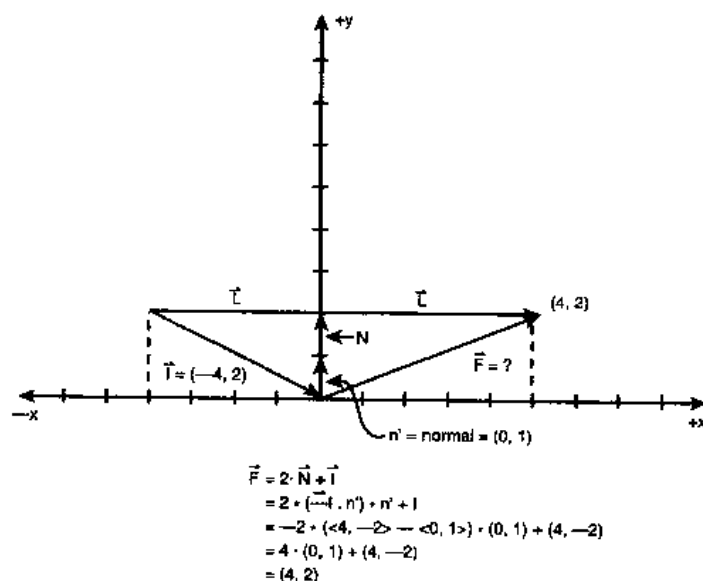


图 13.26 矢量反射的数字示例

物体的初始速率矢量是 $\mathbf{I}=\langle 4, -2 \rangle$, $\mathbf{N}'=\langle 0, 1 \rangle$, 我们需要得到 \mathbf{F} 。把值代入公式:

$$\begin{aligned}\mathbf{F} &= 2 \times \mathbf{N} + \mathbf{I} \\ &= 2 \times (-\mathbf{I} \cdot \mathbf{N}') \times \mathbf{N}' + \mathbf{I} \\ &= -2 \times (\langle 4, -2 \rangle \cdot \langle 0, 1 \rangle) \times \langle 0, 1 \rangle + \langle 4, -2 \rangle \\ &= 4 \times \langle 0, 1 \rangle + \langle 4, -2 \rangle \\ &= \langle 0, 4 \rangle + \langle 4, -2 \rangle \\ &= \langle 4, 2 \rangle\end{aligned}$$

如果你认真看看图 13.26, 那就有正确答案! 现在, 我们只忽略了一个细节问题: 确定什么时候球或物体击中平面或线。

线段的交点

你可能想把这个问题用图表示出来, 我可以帮助你。这个问题基本上是一个线段相交的计算。

但令人吃惊的是现在只是相交线段, 不是直线, 这就是和前面的差别。直线在两端都没有限制, 而线段在两端都有限制, 如图 13.27 所示。

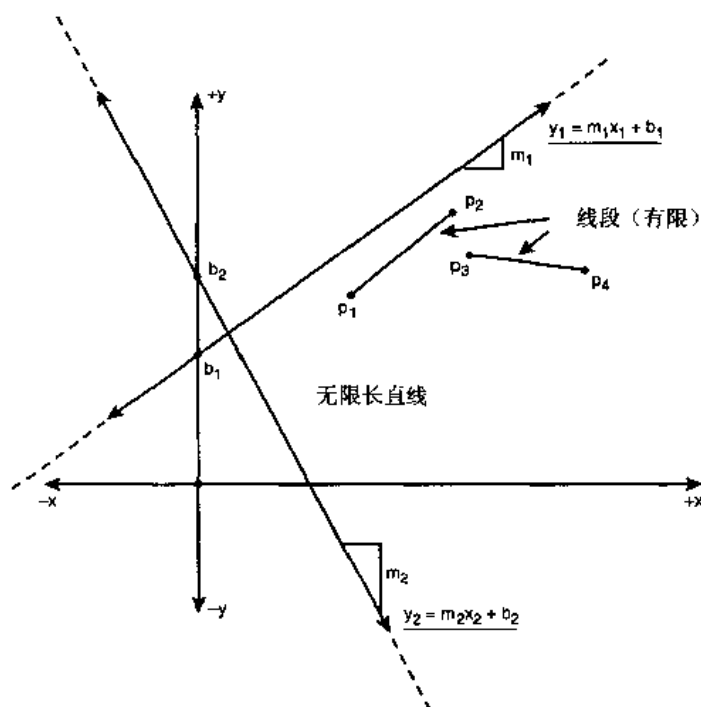


图 13.27 直线和线段是不同的

这个问题可以简化为: 一个运动的物体的速度矢量为 \mathbf{V}_i , 我们想测试物体是否能够穿过碰撞平面或线。如果物体的速率为 \mathbf{V}_i , 那么一帧或一单位时间过后, 物体所处的位置为 $(x_0, y_0) + \mathbf{V}_i$, 或用分量表示:

$$x_1 = x_0 + x_{ix}$$

$$y_1 = y_0 + y_{iy}$$

因此，你可以把速率矢量看作一条引导（我们所画的）物体运动的路线。换句话说，我们想要确定是否有一个线段的交点 (x,y) 。下面是设置：

物体矢量线段： $S_1 = \langle p_1(x_1, y_1) - p_0(x_0, y_0) \rangle$

边界线段： $S_2 = \langle p_3(x_3, y_3) - p_2(x_2, y_2) \rangle$

你需要得到一个准确的交点 (x,y) ，所以当你计算反射矢量 F 时，应该把 F 的初始位置定在 (x,y) 上。如图 13.28 所示。这个问题看起来很简单，但实际上并不像你想的那样容易。因为虽然这些线段是线，但它们的长度是受限制的，所以即使线段的延长线可能相交，但线段却不一定相交。如图 13.29 所示。因此，你不仅需要确定在线段在哪相交，而且你需要确定这个交点是否不在两个线段上。这相对要难一些。

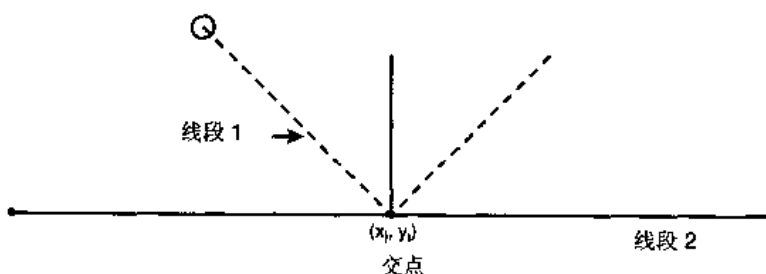


图 13.28 相交和反射

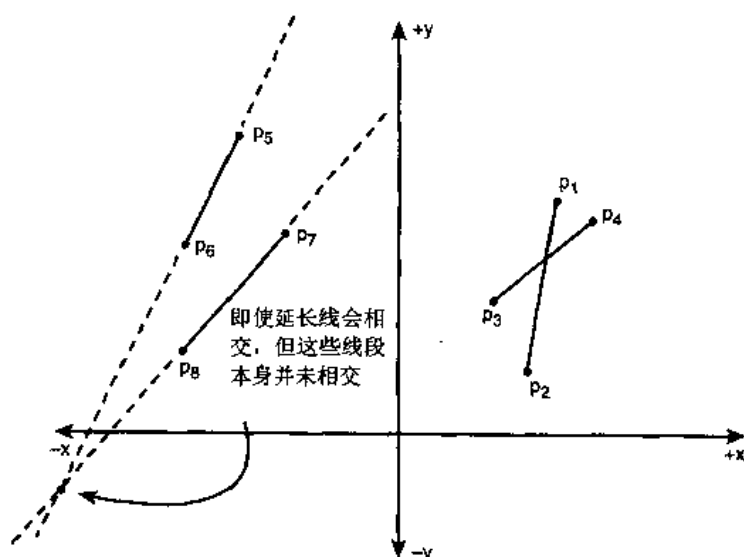


图 13.29 相交的和没有相交的线段

解决这个问题的窍门就是每条线段都用一个参数来代替。我假定 U 为 S_1 上任意一点的位置矢量, V 为 S_2 上任意一点的位置矢量:

$$\text{公式 1: } U = p_0 + t \times S_1$$

$$\text{公式 2: } V = p_2 + s \times S_2$$

约束条件: $(0 \leq t \leq 1), (0 \leq s \leq 1)$ 如图 13.30 所示。

图中我们可以看到, 当 t 从 0 到 1 时, 从 p_0 到 p_1 的线段就会描绘出来。同样, 当 s 从 0 到 1 时, 从 p_2 到 p_3 的线段也会描绘出来。现在我们来解决这个问题。我们用公式 1、2 得出 t 、 s 。把得到的值代入这两个公式的任意一个中去, 得到交点 (x, y) 。而且, 如果我们发现 (s, t) 中任意一个没有在 $(0, 1)$ 的范围内, 那么我们就知道交点没有在线段上。我不准备对完整的推导过程进行很详细的介绍, 因为这个推导过程在很多数学书中都可以找到, 我只给出其中的关键部分。

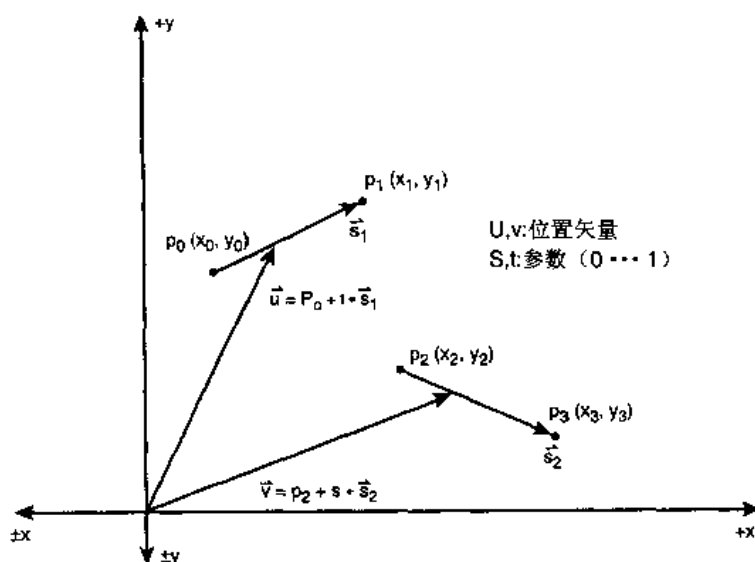


图 13.30 U 和 V 的参数表示

假设:

$$U = p_0 + t \times S_1$$

$$V = p_2 + s \times S_2$$

当 $U = V$, 解出 (s, t) ,

$$p_0 + t \times S_1 = p_2 + s \times S_2$$

$$s \times S_2 - t \times S_1 = p_0 - p_2$$

把上一个公式分解成 (x, y) 分量:

$$s \times S_{2x} - t \times S_{1x} = p_{0x} - p_{2x}$$

$$s \times S_{2y} - t \times S_{1y} = p_{0y} - p_{2y}$$

现在是两个方程式, 两个未知数, 放到矩阵中, 解出 (s, t) :

$$|S_{2x} - S_{1x}| |s| = |(p_{0x} - p_{2x})|$$

$$|S_{2y} - S_{1y}| |s| = |(p_{0y} - p_{2y})|$$

$$\mathbf{A} \quad \mathbf{X} = \quad \mathbf{B}$$

利用克莱姆法则得到:

$$s = \frac{\text{Det} \begin{vmatrix} (p_{0x} - p_{2x}) & -S_{1x} \\ (p_{0y} - p_{2y}) & -S_{1y} \end{vmatrix}}{\text{Det} \begin{vmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{vmatrix}} \quad t = \frac{\text{Det} \begin{vmatrix} S_{2x} & (p_{0x} - p_{2x}) \\ S_{2y} & (p_{0y} - p_{2y}) \end{vmatrix}}{\text{Det} \begin{vmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{vmatrix}}$$

数 学

克莱姆法则是说你可以通过计算 $x_i = \text{Det}(A_i) / \text{Det}$ 解出方程组。 A_i 是用 B 取代 A 的第 i 个的矩。

数 学

通常, 一个矩阵的行列式是相当复杂的。但对于 2×2 或 3×3 的矩阵是很容易记住的。给定一个 2×2 矩阵, 行列式可以这样计算出来:

$$A = \begin{vmatrix} a & b \\ c & d \end{vmatrix} \quad \text{Det} = (a \times d - c \times b)$$

把所有的行列式乘出, 得到

$$s = (-S_{1y} \times (p_{0x} - p_{2x}) + S_{1x} \times (p_{0y} - p_{2y})) / (-S_{2x} \times S_{1y} + S_{1x} \times S_{2y})$$

$$t = (-S_{2x} \times (p_{0y} - p_{2y}) - S_{1y} \times (p_{0x} - p_{2x})) / (-S_{2x} \times S_{1y} + S_{1x} \times S_{2y})$$

得出 s 、 t 后, 可以把任意一个值代入

$$\mathbf{U} = \mathbf{p}_0 + t \times \mathbf{S}_1$$

$$\mathbf{V} = \mathbf{p}_2 + s \times \mathbf{S}_2$$

解出 $U(x,y)$ 或 $V(x,y)$ 。然而, s 、 t 必须在 $[0, 1]$ 范围内才有效。任意一个没有在这个范围内, 就没有交点。参看图 13.31, 我们来看看是否可以用数学方法得到答案。

$$p_0 = (4, 7), p_1 = (16, 3), S_1 = p_1 - p_0 < 12, -4 >$$

$$p_2 = (1, 1), p_3 = (17, 10), S_2 = p_3 - p_2 < 16, 9 >$$

我们知道:

$$s = (-S_{1y} \times (p_{0x} - p_{2x}) + S_{1x} \times (p_{0y} - p_{2y})) / (-S_{2x} \times S_{1y} + S_{1x} \times S_{2y})$$

$$t = (-S_{2x} \times (p_{0y} - p_{2y}) - S_{1y} \times (p_{0x} - p_{2x})) / (-S_{2x} \times S_{1y} + S_{1x} \times S_{2y})$$

代入所有的值, 得到:

$$s = (4 \times (4 - 1) + 2 \times (7 - 1)) / (17 \times 4 + 12 \times 10) = 0.44$$

$$t = (17 \times (7 - 1) - 10 \times (4 - 1)) / (17 \times 4 + 12 \times 10) = 0.383$$

由于 $0 \leq (s, t) \leq 1$, 所以我们知道我们有一个有效交点, 因此 s 和 t 都可以用来求解交点 (x, y) 。我们用 t 。

$$\mathbf{U}(x, y) = \mathbf{p}_0 + t \times \mathbf{S}_1$$

$$= < 7, 7 > + t \times < 12, -4 >$$

代入 $t=0.44$, 得到:

$$= \langle 7, 7 \rangle + 0.44 \times \langle 12, -4 \rangle = \langle 9.28, 5.24 \rangle$$

这就是真正的交点。

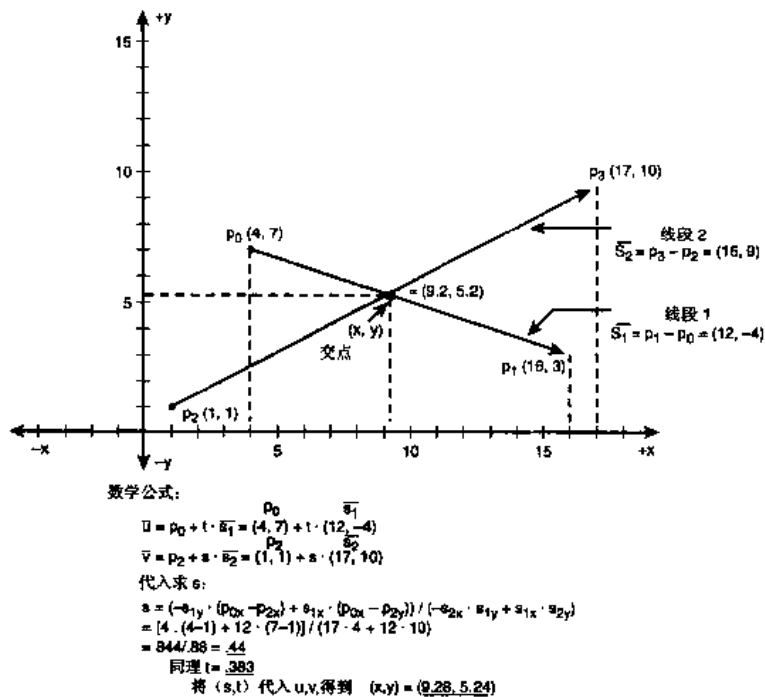


图 13.31 线段相交

我创建了一个演示，演示中使用了这些所有的技术。演示为一个球在一个不规则形状多边形里面弹跳，参见 DEMO13_17.CPP\EXE。如图 13.32 所示。试试调整编码，改变多边形的形状。

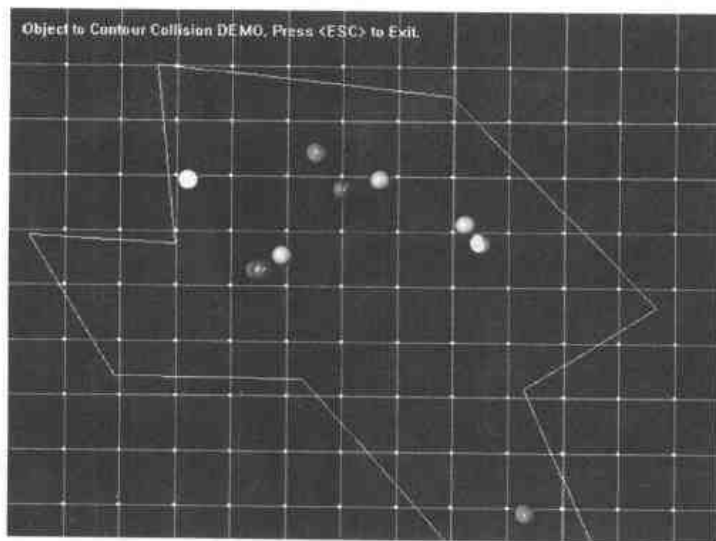


图 13.32 限制在不规则多边形中反弹球

技巧



如果线段的边界框没有重叠，就没有必要测试它们的交点。

最后，当发现一个碰撞轨迹矢量，你可能想用另一种方法试探。在前面的例子中，我们用速率矢量作为测试段。然而，创建一个矢量，其长度等于球的半径，然后让这个矢量从中垂线落到测试边界。这样可以得到比较严格的碰撞，但这更为复杂，把这当作业做做。

2D 物体间的碰撞响应（高级）

我之所以推迟讲述该部分，就是因为我想让你得到一个真正的关于动力和碰撞问题的解决办法，而且要用二者进行数学计算。但是像 Dr.Brown 在《Back to the future》一书中所说：“路？我们在朝哪里走，我们根本就不需要路……”让我们开始吧！

图 13.33 描述了我们想解决的问题。其中有两个通过 2D 环型或 3D 球状模型化的物体，每个都有质量和初始轨线。当它们碰撞之后我们就想计算出最终的轨迹或速度。我们已经在“线性动量的物理性质：守恒和传递”部分接触过这一点了，那是在提出下面的方程时：

线性动量守恒：

$$m_a \mathbf{V}_{ai} + m_b \mathbf{V}_{bi} = m_a \mathbf{V}_{af} + m_b \mathbf{V}_{bf}$$

动能守恒：

$$\frac{1}{2} m_a \mathbf{V}_{ai}^2 + \frac{1}{2} m_b \mathbf{V}_{bi}^2 = \frac{1}{2} m_a \mathbf{V}_{af}^2 + \frac{1}{2} m_b \mathbf{V}_{bf}^2$$

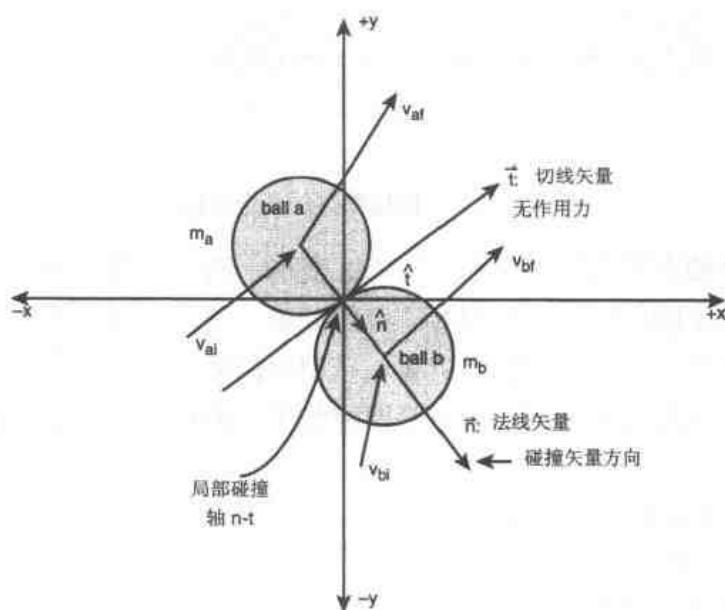


图 13.33 两球体的中心碰撞问题

联合两个方程解出最后的速度，我们可以得到

$$V_{af} = (2m_b V_{bi} + V_{ai} (m_a - m_b)) / (m_a + m_b)$$

$$V_{bf} = (2m_a V_{ai} + V_{bi} (m_a - m_b)) / (m_a + m_b)$$

这些方程对于完全弹性碰撞是正确的。然而却有一个问题，那就是它们是一维的。我们所要做的就是拿出二维问题（像撞球台这样的东西）的解决方法，这有一些复杂。让我们从我们所知道的开始。

我们知道，每一个球（2D 代表）有一些质量 m ；而且两个球全部是由同种材料制成，其质心就是球体的中心。接着，我们知道当两个实际的球相互撞击时，两个球会瞬间产生变形，然后分开。这就是所谓碰撞事件，在图 13.34 中示出。

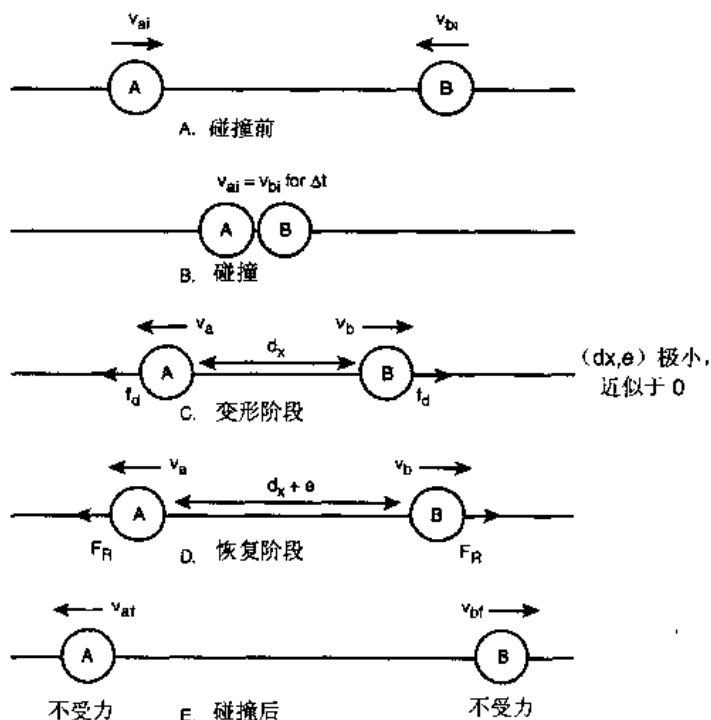


图 13.34 碰撞事件的几个阶段

碰撞事件由两个独立阶段组成。第一阶段：变形，两个球第一次接触并且以相同的速度移动时发生。在变形阶段结束时是恢复阶段的开始，并且一直持续到两个球分开。底线表示在碰撞事件发生时许多我们不能用计算机模型化的复杂物理现象，因此我们必须对碰撞做一些假设。有一点我们不用顾忌，就是即使作了一些假设，在模拟时它也看起来特别逼真。所做的假设如下：

1. 碰撞的时间非常短，记为 dt 。
2. 碰撞时球的位置并不动。
3. 球速可以做显著的改变。
4. 在碰撞时没有摩擦力作用。

假设 3 是我惟一需要阐明的, 因为我觉得其他几条都很容易理解。要想让假设 3 成立, 一个瞬间力必须在碰撞时加入。这个力被称做推动力。这是解决问题的关键。当球相撞时就会产生一个巨大短时的力——推动力。我们可以计算出推动力, 而且从结果中可以提出另一个方程来帮助解决 2D 问题。这里要用高等数学的微积分知识, 所以我要略过它。结果是碰撞事件时模拟所有物理现象系数的产生:

方程1: 回归系数

$$e = \frac{V_{bf} - V_{af}}{V_{bi} - V_{ai}}$$

方程 1 可以求得 e , 它称为回归系数。它在碰撞之前和之后模拟速度和损失的动能。如果你设定 e 为 1 那么模拟就是完全弹性碰撞。换句话说, 如果 $e < 1$ 的话, 模拟非完全弹性碰撞, 而且碰撞后的球速和线性动量都会有所损失。现在问题是在哪里可以得到 e , 这需要设定或查阅。有趣的是, 如果你为了 e 而联合动量守恒方程:

$$m_a V_{ai} + m_b V_{bi} = m_a V_{af} + m_b V_{bf}$$

你就会得到下面的结果:

方程2: 最终的速率

$$V_{af} = ((e+1) m_b V_{bi} + V_{ai}(m_a - m_b)) / (m_a + m_b)$$

$$V_{bf} = ((e+1) m_a V_{ai} - V_{bi}(m_a - m_b)) / (m_a + m_b)$$

难道不有趣吗? 几乎等同于我们联解动能方程与线性动量方程时得出的公式。并且事实上我们在联解动能方程与线性动量方程时所做的假设是动能被保存了下来。如果我们现在设定 $e=1$, 我们就会得到:

$$V_{af} = ((e+1) m_b V_{bi} + V_{ai}(m_a - m_b)) / (m_a + m_b)$$

$$V_{bf} = ((e+1) m_a V_{ai} - V_{bi}(m_a - m_b)) / (m_a + m_b)$$

或

$$V_{af} = (2m_b V_{bi} + V_{ai}(m_a - m_b)) / (m_a + m_b)$$

$$V_{bf} = (2m_a V_{ai} - V_{bi}(m_a - m_b)) / (m_a + m_b)$$

这些确实是带有动能和线性动量存储的方程。看来我们走对了。由方程 1 和 2 可以解决这个问题。但是关键的一点是方程仍然是针对 1D 的, 所以我们需要写出针对 2D 的方程, 然后找到解法。

返回到图 13.33, 你可以看到有两个标记着 n 和 t 的附加轴。 n 轴是碰撞线的方向, t 轴或切线轴垂直于 n 轴。假设我们已经计算出代表这些轴的向量 (稍后我会演示), 那么就可以写出一些方程。

我们要写出的方程的第一个设定同碰撞前后的速度切线构成有关。因为没有摩擦力和外力作用, 在碰撞的切线方向 (相信我) 切向的线性动量 (和速度) 前后一定是相同的, 对吗? 如果没有力, 这一点一定是正确的, 这样我们就可以写出方程 3: 最初与最后的切向动量/速度间的关系。

方程3:

$$m_a(V_{ai})t = m_a(V_{af})t$$

$$m_b(V_{bi})t = m_b(V_{bf})t$$

如果你愿意, 可以合起来这样写:

$$m_a(V_{ai})t + m_b(V_{bi})t = m_a(V_{af})t + m_b(V_{bf})t$$

数 学

我的想法很简单; (a,b) 代表球, (i,f) 代表开始或最后, (n,t) 代表沿着n或t的轴。

既然质量在碰撞前后是相同的, 我们就可以消掉质量, 从而得出速度也是相同的:

方程4: 碰撞后的速度等同于切向速度

$$(V_{ai})t = (V_{af})t$$

$$(V_{bi})t = (V_{bf})t$$

既然我们已经解决了问题的一半, 我们就知道了切向的最后速度。让我们找一下正常构成或碰撞线 n 方向的速度。我们知道线性动量总是被存储, 因为碰撞时没有外力作用在球上, 所以我们可以写出:

方程5: 线性动量存储在n轴或碰撞线上

$$m_a(V_{ai})n + m_b(V_{bi})n = m_a(V_{af})n + m_b(V_{bf})n$$

根据n我们也可以写出e:

方程6: n轴的回归系数:

$$e = \frac{(V_{bf})n - (V_{af})n}{(V_{bi})n - (V_{ai})n}$$

现在让我们看一看我们得到的公式。如果你看下方程5和6, 我突出显示了我们没有的变量: $(v_{af})n$ 和 $(v_{bf})n$, 只是最后速度的法线分量。噢! 我们有两个方程和两个未知数, 所以可以将其解出来。但我们已经有了答案! 方程2仍是代表某个特别的轴, 所以我再写一遍沿n向的分量:

方程7: 法线方向的最后速度

$$V_{af} = ((e+1) m_b V_{bin} + V_{ai}(m_a - e m_b)) / (m_a + m_b)$$

$$V_{bf} = ((e+1) m_a V_{ain} - V_{bin}(m_a - e m_b)) / (m_a + m_b)$$

解决 n-t 坐标系

既然我们已经有了最后碰撞的响应, 我们就需要搞清楚如何获取 $(V_{ai})n$ 、 $(V_{ai})t$ 、 $(V_{bi})n$ 、 $(V_{bi})t$ 的初始值, 问题解决后我们又必须把n-t轴中的值转换回到x、y轴中。让我们首先从找到向量n和t开始。

要找到n我们就需要一个向量, 它是单位长度 (长度等于1.0), 而且方向沿着球A(x_{a0} , y_{a0})

的中心到球B (x_{b0} , y_{b0}) 的中心。让我们通过找到一个球A到B的向量开始, 把它叫做 \mathbf{N} , 然后归整如下:

方程8: \mathbf{n} 和 \mathbf{t} 的计算

$$\mathbf{N} = \mathbf{B} - \mathbf{A} = \langle x_{b0} - x_{a0}, y_{b0} - y_{a0} \rangle$$

归整 \mathbf{N} 来找到 \mathbf{n} , 我们可以得到

$$\mathbf{n} = \mathbf{N} / |\mathbf{N}| = \langle n_x, n_y \rangle$$

现在我们需要垂直于 \mathbf{n} 的切向轴 \mathbf{t} 。我们可以再一次使用向量几何来找到它, 但也可以使用一个技巧: 如果我们把 \mathbf{n} 顺时针旋转90度就是我们想要的向量。如果一个2D向量 $\langle x, y \rangle$ 在一个平面上顺时针旋转90度, 那么旋转后的向量就是 $\mathbf{t} = \langle -y, x \rangle = \langle t_x, t_y \rangle$ 。看一看图13.35。

既然我们有了 \mathbf{n} 和 \mathbf{t} , 而且它们都是单位向量 (因为 \mathbf{n} 是单位的, \mathbf{t} 也是单位的), 我们就准备开始吧。对于A和B, 根据 \mathbf{n} 和 \mathbf{t} 解决两个球的初速度, 分别为:

$$\mathbf{V}_{ai} = \langle x_{vai}, y_{vai} \rangle$$

$$\mathbf{V}_{bi} = \langle x_{vbi}, y_{vbi} \rangle$$

这只是点积。要找到 $(\mathbf{V}_{ai})\mathbf{n}$ 即球A沿 \mathbf{n} 轴的初速度, 可以这样做:

$$\begin{aligned} (\mathbf{V}_{ai})\mathbf{n} &= \mathbf{V}_{ai}\mathbf{n} = \langle x_{vai}, y_{vai} \rangle \cdot \langle n_x, n_y \rangle \\ &= (x_{vai} \times n_x + y_{vai} \times n_y) \end{aligned}$$

记住该结果是个标量。同样在方程9中进行其他初速度的计算。

方程9: \mathbf{V}_{ai} 沿 \mathbf{n} 和 \mathbf{t} 的速度分量:

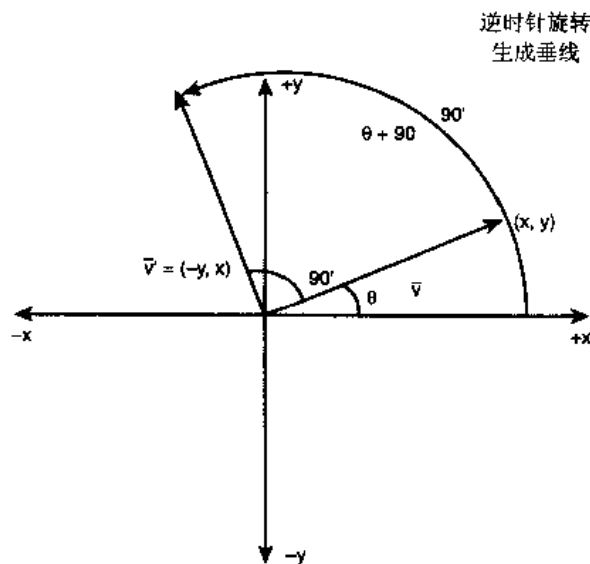


图 13.35 旋转向量 90° 求得正交向量

$$(\mathbf{V}_{ai})\mathbf{n} = \mathbf{V}_{ai}\mathbf{n} = \langle x_{vai}, y_{vai} \rangle \cdot \langle n_x, n_y \rangle$$

$$= (x_{vai} \times n_x + y_{vai} \times n_y)$$

$$(\mathbf{V}_{ai})\mathbf{t} = \mathbf{V}_{ai}\mathbf{t} = \langle x_{vai}, y_{vai} \rangle \cdot \langle t_x, t_y \rangle$$

很明显,我们要把 \mathbf{n}, \mathbf{t} 坐标系统中的一个向量转换到 x, y 坐标系统。但是怎样作呢?再一次,我们打算使用点积。看一看图13.36中的向量 $(V_{af})_{nt}$, 不考虑 \mathbf{n}, \mathbf{t} 轴。我们只想知道 x, y 系统中的 $(V_{af})_{nt}$ 。要计算它,我们需要沿 x 和 y 轴分解 $(V_{af})_{nt}$, 用点积可以发现这一点。所有我们需要作的就是下面的点积:

球A在 \mathbf{n}, \mathbf{t} 系统中的 V_a

$$V_a = (V_{af})\mathbf{n} \times \mathbf{n} + (V_{at})\mathbf{t} \times \mathbf{t}$$

$$(V_{af})\mathbf{n} \times \langle n_x, n_y \rangle + (V_{at})\mathbf{t} \times \langle t_x, t_y \rangle$$

所以,写成点积形式

$$x_{af} = \langle n_x, 0 \rangle \cdot (V_{af})\mathbf{n} + \langle t_x, 0 \rangle \cdot (V_{at})\mathbf{t}$$

$$= n_x \times (V_{af})\mathbf{n} + t_x \times (V_{at})\mathbf{t}$$

$$y_{af} = \langle n_y, 0 \rangle \cdot (V_{af})\mathbf{n} + \langle t_y, 0 \rangle \cdot (V_{at})\mathbf{t}$$

$$= n_y \times (V_{af})\mathbf{n} + t_y \times (V_{at})\mathbf{t}$$

对于球B:

$$V_b = (V_{bf})\mathbf{n} \times \mathbf{n} + (V_{bt})\mathbf{t} \times \mathbf{t}$$

$$(V_{bf})\mathbf{n} \times \langle n_x, n_y \rangle + (V_{bt})\mathbf{t} \times \langle t_x, t_y \rangle$$

写成点积形式

$$x_{bf} = \langle n_x, 0 \rangle \cdot (V_{bf})\mathbf{n} + \langle t_x, 0 \rangle \cdot (V_{bt})\mathbf{t}$$

$$= n_x \times (V_{bf})\mathbf{n} + t_x \times (V_{bt})\mathbf{t}$$

$$y_{bf} = \langle n_y, 0 \rangle \cdot (V_{bf})\mathbf{n} + \langle t_y, 0 \rangle \cdot (V_{bt})\mathbf{t}$$

$$= n_y \times (V_{bf})\mathbf{n} + t_y \times (V_{bt})\mathbf{t}$$

用上面的速度把球发射出去,你就完成了。现在,因为这些代码比数学推导容易理解得多,所以我就列出即将演示的碰撞运算法则:

```
void Collision_Response(void)
{
    //this function does all the "real " physics to determine if there has
    // been a collision between any ball and any other ball; if there is a
    //collision ,the function uses the mass of each ball along with the
    // initial velocities to compute the resulting velocities
    //from the book we know that in general
    //va2= (e+1)*mb*vb1+va1(ma-e*mb)/(ma+mb)
    //vb2= (e+1)*ma*vba+vb1(mb-e*ma)/(mb+ma)
    //and the objects will have direction vectors co-linear to the normal
    //of the pointe of collision ,but since we are suing spheres here as the
    //objects, we know that the normal to the point of collision is just
    //the vector from the centers of each object, thus the resulting
    //velocity vector of each ball will be along this normal vector direction
    //step 1:test each object against each other object and test for a
    //collision;there are better ways to do this other than a double nested
    //loop,but since there are a small number of bojects this is fine;
    //also we want to somewhat model if two or more balls hit sumultaneously
```

```

for( int ball_a=0;ball_a<NUM_BALLS;ball_a++)
{
    for( int ball_b=0;ball_b<NUM_BALLS;ball_b++)
    {
        if (ball_a==ball_b)
            continue;
        //compute the normal vector from a->b
        float nabx=(balls[ball_b].varsF[INDEX_X]-
            balls[ball_a].varsF[INDEX_X]);
        float naby=(balls[ball_b].varsF[INDEX_Y]-
            balls[ball_a].varsF[INDEX_Y]);
        float length=sqrt(nabx*nabx+naby*naby);
        //is there a collision?
        if (length<=2.0*(BALL_RADIUS*.75))
        {
            //the balls have made contact,compute response
            //compute the response coordinate system axes
            //normalize normal vector
            nabx/=length;
            naby/=length;
            //compute the tangential vector perpendicular to normal,
            //simply rotate vector 90
            float tabx=-naby;
            float taby=nabx;
            //draw collision
            Ddraw_Lock_Primary_Surface();
            //blue is normal
            Draw_Clip_Line(balls[ball_a].varsF[INDEX_X]+0.5,
                balls[ball_a].varsF[INDEX_Y]+0.5
                balls[ball_a].varsF[INDEX_X]+20*naby+0.5,
                balls[ball_a].varsF[INDEX_Y]+20*naby+0.5,
                252,primary_buffer,primary_lpitch);
            //yellow is tangential
            Draw_Clip_Line(balls[ball_a].varsF[INDEX_X]+0.5,
                balls[ball_a].varsF[INDEX_Y]+0.5
                balls[ball_a].varsF[INDEX_X]+20*taby+0.5,
                balls[ball_a].varsF[INDEX_Y]+20*taby+0.5,
                251,primary_buffer,primary_lpitch);
            Ddraw_Unlock_Primary_Surface();
            // tangential is also normalized since
            //it's just a rotated normal vector
            //step2:compute all the initial velocities
            //notation ball(a,b) initial :i,final:f,
            //n:normal direction, t:tangential direction
            float vait=DOT_PRODUCT(balls[ball_a].varsF[INDEX_XV],
                balls[ball_a].varsF[INDEX_YV],
                tabx,taby);
            float vait=DOT_PRODUCT(balls[ball_a].varsF[INDEX_XV],
                balls[ball_a].varsF[INDEX_YV],

```

```

        nabx,naby);
float vait=DOT_PRODUCT(balls[ball_b].varsF[INDEX_XV],
        balls[ball_b].varsF[INDEX_YV],
        tabx,taby);
float vait=DOT_PRODUCT(balls[ball_b].varsF[INDEX_XV],
        balls[ball_b].varsF[INDEX_YV],
        nabx,naby);
// now we have all the initial velocities
//in terms of the n and t axes
//step 3:compute final velocities after
//collision ,from book we have
//note: all this code be optimized, but I want you
// to see what's happening)
float ma =balls[ball_a].varsF[INDEX_MASS];
float mb =balls[ball_b].varsF[INDEX_MASS];
float vafn=(mb*vbin*(cof_E+1)+vain*(ma-cof_E*mb))
        /(ma+mb);
float vbfn=(mb*vain*(cof_E+1)+vbin*(ma-cof_E*mb))
        /(ma+mb);
//now luckily the tangential components
//are the same before and after ,so
float vaft=vait;
float vbft=vbit;
// and that's that baby!
// the velocity vectors are:
//object a(vafn,vaft)
//object b(vbfn,vbft)
//the only problem is that we are in the wrong coordinate
//system! We need to //translate back to the original x,y
//coordinate system; basically we need to
//compute the sum of the x components relative to
// the n,t axes and the sum of
// the y components relative to the n,t axis,
// since n,t may both have x,y
//components in the original x,y coordinate system

float xfa=vafn*nabx+vaft*tabx;
float yfa=vafn*naby+vaft*taby;

float xfb=vbfn*nabx+vbft*tabx;
float yfb=vbfn*naby+vbft*taby;
// store results
balls[ball_a].varsF[INDEX_XV]=xfa;
balls[ball_a].varsF[INDEX_YV]=yfa;

balls[ball_b].varsF[INDEX_XV]=xfb;
balls[ball_b].varsF[INDEX_YV]=yfb;

// update position

```

```

balls[ball_a].varsF[INDEX_X] +=
    balls[ball_a].varsF[INDEX_XV];
balls[ball_a].varsF[INDEX_Y] +=
    balls[ball_a].varsF[INDEX_YV];
balls[ball_b].varsF[INDEX_X] +=
    balls[ball_b].varsF[INDEX_XV];
balls[ball_b].varsF[INDEX_Y] +=
    balls[ball_b].varsF[INDEX_YV];
    }// end if
    }// end for ball2
} //end for ball1
} // end Collision_Response

```

代码几乎完全等同于运算法则。然而代码来自一个模拟撞球台系统的演示程序，所以每一对碰撞都增加了循环检测。循环代码内部遵循数学法则。要看运算法则的运行可以检测一下程序DEMO13.CPPIEXE。图13.37显示了其中的一个画面的快照。演示程序是以一些随机移动的球开始的，然后物理学开始起作用。屏幕的低端显示了总的能量值。通过左右方向键试图改变一下回归系数，看看有什么现象发生。如果值小于1，系统的能量就会减少，如果值等于1，系统的能量不变，如果值大于1，系统的能量就会增加——我希望我的银行账户就是如此。

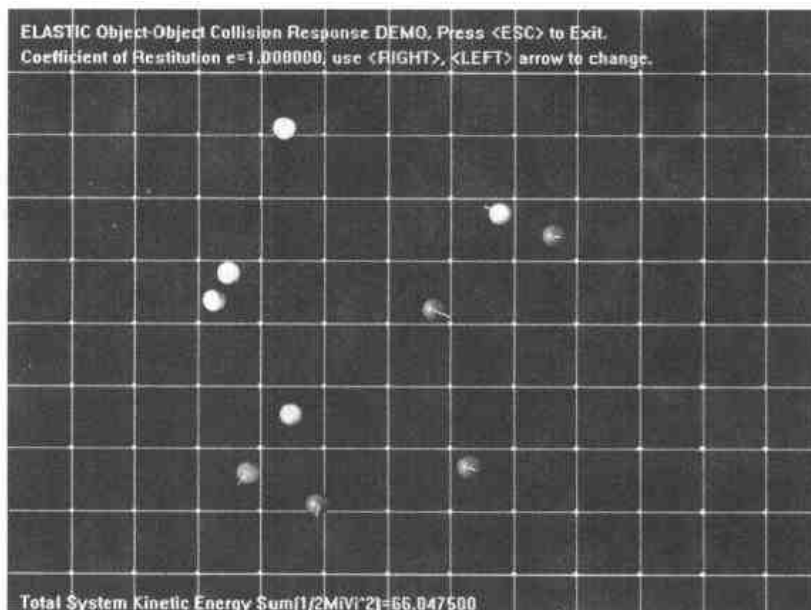


图 13.37 超级碰撞响应模型

简单运动学

运动学这个词的含义十分丰富。对3D艺术家来说它代表一件事情，对3D游戏程序它代

表着另一件事情，而对于物理学来说，它的含义就又不同了。然而，在本书的这一部分，它的含义是刚体的连续运动力学。在计算机时代还有两个运动学问题。第一个是正向运动学问题，第二个是逆向运动学问题。图13.38显示出了正向运动学问题：你可以看到一个2D连续刚体（直臂）。每一个点可以在平面上自由旋转，所以在该例子中都有两个自由度 θ_1 和 θ_2 。而且，每一个臂都有长度 L_1 和 L_2 。正向运动学可以这样开始：给出 θ_1 、 θ_2 、 L_1 、 L_2 ，找出 p_2 的位置。

为什么我们会对这个感兴趣呢？好，如果你打算写一个2D或3D的游戏，并且拥有联接四周运动的实时模型，你最好知道怎样做到这一点。例如，3D动画是通过两种方式达到的。最快而最笨的方法是拥有一套代表3D动画物体的网点。较灵活的方法是有一个3D网点，该点有许多结合点和臂，然后通过3D模型来“运行”动作数据。然而要作到这一点你必须理解移动手臂时涉及到腕、肘、肩及臀部等的物理/机械特点——明白了吗？

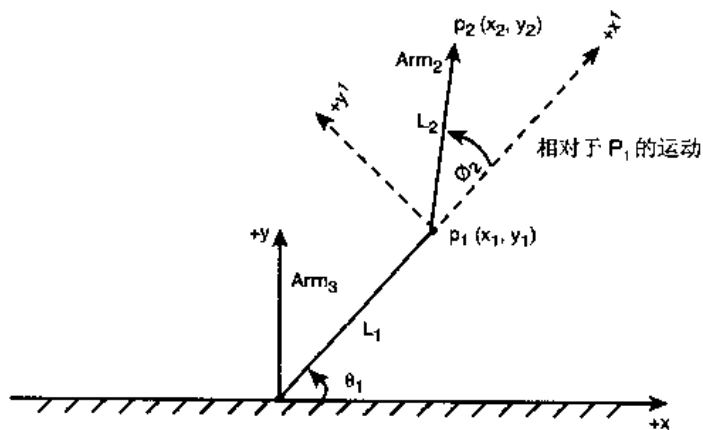


图 13.38 正向运动学问题

第二个运动学问题是第一个的逆向：

给出位置 p ，找出物理模型中所有满足 L_1 和 L_2 的 θ_1 和 θ_2 的值。这比你所能想像的要复杂得多。图13.39显示出了这一点。

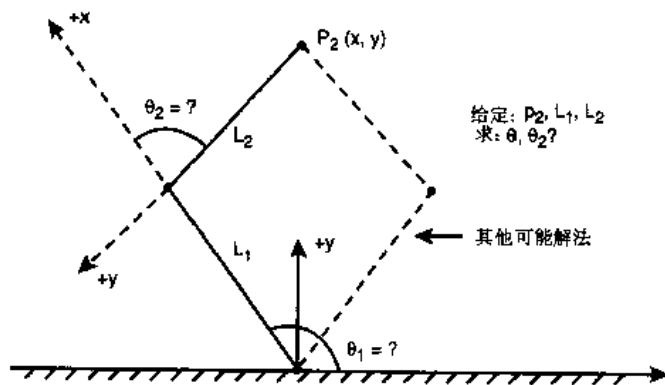


图 13.39 逆向运动学问题

参考该图，你可以看到有两个可能的方法满足所有的约束。我打算简单地处理一下这个问题，因为大多数情况下我们不需要解决这个问题，数学上也是很粗糙的，但是稍后我会给出一个例子说明怎样做。

解决正向运动学问题

我想做的就是给你演示如何解决正向运动学问题，因为这很容易作到。回到图13.38，问题不过是相对运动。如果你相对于节点2看待此问题，那么定位 p_2 不过是对 l_2 的平移和对 θ_2 的旋转。然而，点 p_2 本身也仅仅是通过 p_1 的 l_1 的平移和 θ_1 的旋转得到定位。因此，问题解决的方法不过是框架到框架、连接到连接的平移和旋转。让我们分开解决这个问题。

忘掉第一个臂，把焦点放在第二个，也就是把我们的工作后退一点。起点是 p_1 ，我们要沿 x -轴方向移动 l_2 ，然后在 x 、 y 平面上旋转 θ_2 （或在3D中围绕 z 轴）来定位 p_2 。这很容易——所有我们需要作的就是从 p_1 点平移：

$$p_2 = p_1 \times T_{l_2} \times R_{\theta_2}$$

但是我们却没有 p_1 ？那好吧——我们假设它已知。不管怎样 T_{l_2} 和 R_{θ_2} 都是你在第八章“矢量光栅化及2D变换”中了解的标准2D平移和旋转。所以我们有

$$T_{l_2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & 0 & 1 \end{bmatrix}, \quad R_{\theta_2} = \begin{bmatrix} \cos \theta_2 & \sin \theta_2 & 0 \\ -\sin \theta_2 & \cos \theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

因此 p_2 就是乘积：

$$p_2 = p_1 \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos \theta_2 & \sin \theta_2 & 0 \\ -\sin \theta_2 & \cos \theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

如果我们可以从 p_1 计算出 p_2 ，那么 p_1 就应该可以从 p_0 计算出来，也就是说通过平移 l_1 和旋转 θ_1 。数学表达为：

$$p_2 = p_0 \times T_{l_2} \times R_{\theta_2} \times R_{\theta_1}$$

这里 p_0 是 $[0,0,1]$ ，同样2D坐标的起点（我们可以使用任何想使用的点，但起码这代表运动学链接的基础）。2D系统中三个分量构成的原因是这样我们可以用矩阵完成转换和平移。所以最后的1.0是位置固定点。所有的点都是在 $(x,y,1)$ 形式中。而且 T_{l_1} 和 R_{θ_1} 与 T_{l_2} 和 R_{θ_2} 一样有同样的形式，但有不同的值。记住乘积的顺序——因为我们在做前面的工作，我们必须首先通过 $T_{l_2} \times R_{\theta_2}$ 然后通过 $T_{l_1} \times R_{\theta_1}$ 转换 p_0 ，所以顺序是有要求的！

把所有的都记在大脑中，我们看到 p_2 确实是通过起点 p_0 乘以矩阵 $(T_{l_2} \times R_{\theta_2}) \times (T_{l_1} \times R_{\theta_1})$ 得到。这可以推广到所有需要的情况，或写为：

$$p_n = p_0 \times T_n \times R_n \times T_{n-1} \times R_{n-1} \times T_{n-2} \times R_{n-2} \times \cdots \times T_1 \times R_1$$

这是 n 个连接点的情况。

这是可行的，因为每一对矩阵的积都转换相连的坐标系统，所以这些转换的积如同是可以定位最后点的坐标转换队列。作为一个例子，让我们看一看这个巨大的工作。图13.40详细地描述了问题解决的曲线说明。

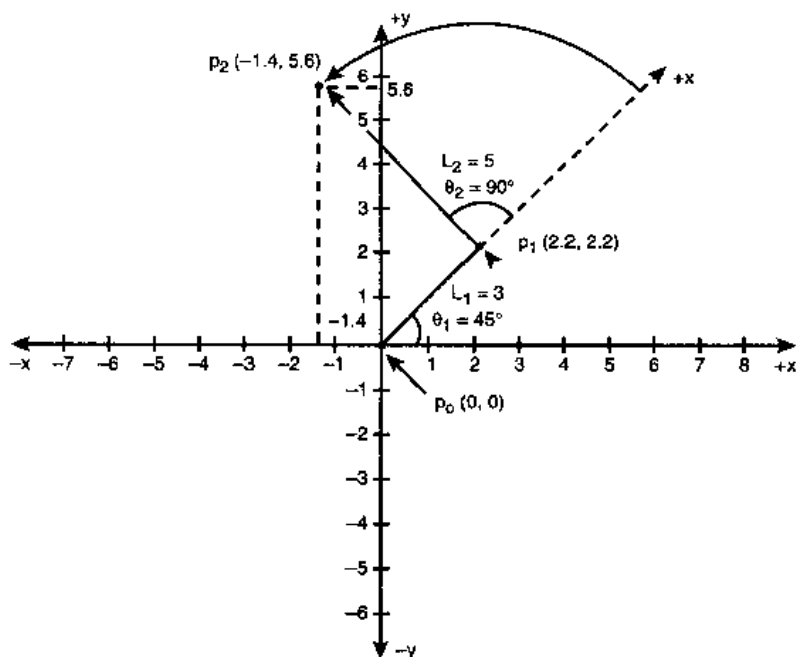


图 13.40 运动学链的图解

我已经标出了点、角度等，用圆规和直尺计算给出输入值的 p_2 的位置：

$$l_1=3, l_2=5$$

$$\theta_1=45^\circ, \theta_2=90^\circ$$

$$p_0=(0,0)$$

从图中我可以粗略地估计出

$$p_2=(-1.4, 5.6)$$

现在让我们看看数学计算是否带给我们同样的结果。

$$p_2 = [0, 0, 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 5 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0.707 & 0.707 & 0 \\ -0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$P_0 \quad T_{12} \quad R_{e2} \quad T_{11} \quad R_{o1}$

$$= [0, 0, 1] \times \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 5 & 1 \end{bmatrix} \times \begin{bmatrix} 0.707 & 0.707 & 0 \\ -0.707 & 0.707 & 0 \\ 2.121 & 2.121 & 1 \end{bmatrix}$$

$P_0 \quad T_{12} \times R_{e2} \quad T_{11} \times R_{o1}$

$$= [0, 0, 1] \times \begin{vmatrix} -0.707 & 0.707 & 0 \\ 0.707 & 0.707 & 0 \\ 1.414 & 5.656 & 1 \end{vmatrix}$$

$$P_0 \quad T_{12} \times R_{02} \times T_{11} \times R_{01}$$

$$p_2 = [-1.414, 5.656, 1]$$

去掉1.0, 因为 $[x, y, 1]$ 实际上代表 $x'=x/1, y'=y/1$, 或 $x'=x, y'=y$, 我们就有:

$$p_2 = (-1.414, 5.656)$$

如果你看图13.40, 那么会发现它非常的接近。这就是2D中正向运动学的全部内容。当然, 由于z轴的存在, 3D中这样做是有一些复杂, 但是只要你挑选一个旋转的协议, 同样可以工作出来。图13.41显示了我创建的程序DEMO13_9.CPP1EXE, 是一个正向运动学的例子。它叫你改变两个联臂的角度, 然后计算出 p_1 和 p_2 的位置并显示出来。A、S、D、F四个键控制臂1和臂2的角度, 看一下你是否可以增加一个约束, 使 p_2 的最后受动器不会落在 $y=0$ 轴以下, 通过蓝线表示。

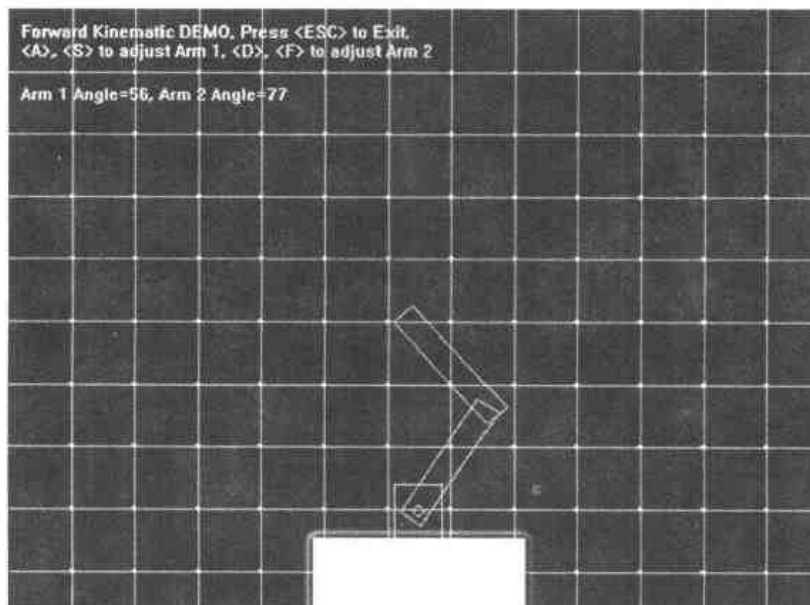


图 13.41 运动学链的演示

解决逆向运动学问题

解决逆向运动学问题通常是相当复杂的, 但我会给你一个感性认识, 以便你可以知道从哪里开始。以前的部分是知道了 P_0 、 l_1 、 l_2 、 θ_1 、 θ_2 来求 p_2 , 但是如果你不知道 θ_1 、 θ_2 而知道 p_2 呢?

该运动学问题的解决方法可以是建立约束方程系统然后解出未知角度。这个问题可以有一个不确定系统也就是说有不止一种解决方法。因此你必须增加其他启发或约束来找到你要解决的问题。

作为例子我们来看一个简单的仅有一个臂的问题，你可以看到过程。图13.42显示了有一个与x轴角度为 θ_1 的臂 l_1 。给出 $p_1(x_1, y_1)$ ， θ_1 是多少？

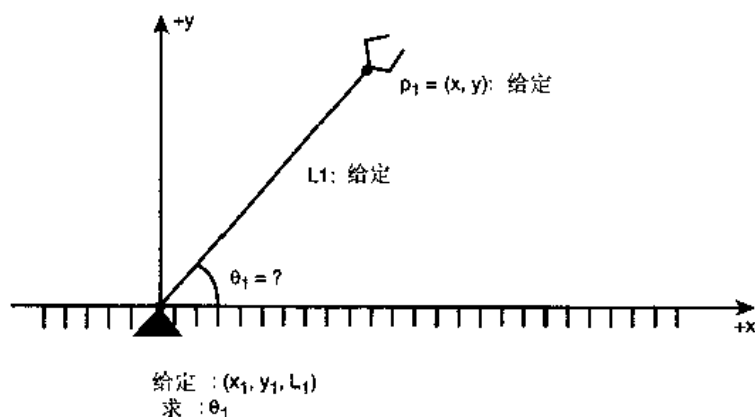


图 13.42 单臂逆向运动学问题

我们可以像这样用正向运动学矩阵来解决这个问题：

$$p_1 = p_0 \times T_{11} \times R_{01}$$

$$p_1 = [0, 0, 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos \theta_1 & \sin \theta_1 & 0 \\ -\sin \theta_1 & \cos \theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$P_0 \quad T_{11} \quad R_{01}$

$$= [l_1, 0, 1] \times \begin{bmatrix} \cos \theta_1 & \sin \theta_1 & 0 \\ -\sin \theta_1 & \cos \theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P_0 \times T_{11} \quad R_{01}$$

$$p_1(x_1, y_1) = (l_1 \times \cos \theta_1, l_1 \times \sin \theta_1, 1)$$

所以，

$$x_1 = l_1 \times \cos \theta_1$$

$$y_1 = l_1 \times \sin \theta_1$$

$$\theta_1 = \cos^{-1} x_1 / l_1$$

或，

$$\theta_1 = \sin^{-1} y_1 / l_1$$

我可以把问题完全用矩阵形式讲解，但这样更有说明性。

OK，这个系统是多种因素决定的；换句话说，一旦你选择了x 或y，那么另一个就通过 θ_1 来决定。这很有趣，但你仔细考虑一下就会发现臂 l_1 使我们丧失了一个自由度，因此你不可以再定位任何你想要的点了，事实上，惟一的有效点的构成为：

$$x_1 = l_1 \times \cos \theta_1$$

$$y_1 = l_1 \times \sin \theta_1$$

如果有两个连接，你就会发现对任意 x, y 而言，都会存在多个 θ_1, θ_2 的值满足方程组。

微粒系统

这是个重要的主题。每个人都在说“有微粒系统吗？”好，微粒系统可能很复杂，也可能很简单。最基本的是微粒系统和可以模拟微粒的物理学模型。对于你的游戏中的爆炸、水气尾迹和通常的光线显示是很重要的。关于物理学模拟你已经了解了很多，我相信你可以创建自己的微粒系统。不过，只是为了让你开个头，我打算让你看看如何创建一个快速且简单的基于像素大小的微粒系统。

假定我们想用微粒来产生爆炸，也可以是水气尾迹。因为一个微粒系统是由 n 个微粒组成，所以让我们把焦点放在一个微粒的模型上。

每个微粒所需要的东西

如果你需要你就可以模拟碰撞响应、动量转换和其他所有的现象，但大多数点系统只有极简单的模型。下面是一个花园类微粒的特征：

- 位置
- 速度
- 颜色/活度
- 生命周期
- 重力
- 风力

当你建立一个微粒时，就至少要给它一个位置、初速度、颜色、生命周期。当然，这个微粒也可以是一颗煤渣，那么就应该包括进颜色活度。而且，你想拥有一些作用在所有微粒上的力，像引力和风。你也可以拥有微粒的创建收集功能，这些是你所最初渴望的条件，比如爆炸或水气尾迹。当然，你也可以让微粒有像实际生活中那样的反弹离物体的能力。然而大多数时间里微粒们都默默无闻，没有人关心它们！

设计一个微粒工程

要设计一个点系统，你需要三个独立的要素：

- 微粒数据结构
- 加工微粒的微粒发动机
- 产生特殊微粒初始条件的功能

让我们从数据结构开始。我要假定是8位显示，因为在活度方面用字节处理比RGB更容

易。不管怎样，下面是单个微粒的首次尝试：

```
//a single particle
typedef struct PARTICLE_TYP
{
    int state;           //state of the particle
    int type;           //type of particle edect
    float x,y;          //world position of particle
    float xv,yv;        //velocity of particle
    int curr_color;      //the current rendering color of particle
    int start_color;     //the start color or range effect
    int end_color;       //the ending color of range effect
    int counter         //general state transition timer
    int max_count;       //max value for counter
}PARTICLE,*PARTICLE_PTR
```

让我们加入一些全局变量来处理外部影响，比如Y方向的引力和X方向的风力。

```
Float particle_wind.0;      //assume it operates in the x direction
Float particle_gravity.0;   //assume it operates in the y direction
```

让我们定义一些常量，这些是我们需要完成的影响中的一些：

```
//defines for particle system
#define PARTICLE_STATE_DEAD      0
#define PARTICLE_STATE_ALIVE    1
//type of particles
#define PARTICLE_TYPE_FLICKER   0
#define PARTICLE_TYPE_FADE      1
//color of particle
#define PARTICLE_COLOR_RED      0
#define PARTICLE_COLOR_GREEN    1
#define PARTICLE_COLOR_BLUE     2
#define PARTICLE_COLOR_WHITE    3
#define MAX_APRTICLES           128
//color ranges (based on my palette)
#define COLOR_RED_START         32
#define COLOR_RED_END           47
#define COLOR_GREEN_START       96
#define COLOR_GREEN_END         111
#define COLOR_BLUE_START        144
#define COLOR_BLUE_END          159
#define COLOR_WHITE_START       16
#define COLOR_WHITE_END         31
```

这里可以明白我的想法了。我想让那个微粒或是红色、绿色、蓝色或是白色，所以我调用一个调色板，同时搞清楚范围内的颜色指数。如果你想使用16位色，那么你就亲手改动RGB的始末值——我将使它简单化。当然，你看到了我正计划制作两种微粒：衰退型和

闪烁型。衰退微粒仅仅逐渐减弱，但闪烁微粒则将闪个不停，像是火花。

最后，让我们来存储它们：

```
PARTICLE particles[MAX_PARTICLES]; //the particles for the particle engine
```

这样就让我们开始写这个函数来控制每个微粒。

微粒引擎软件

我们需要函数来初始化所有的微粒，启动一个微粒，处理所有的微粒以及完成时清除所有的微粒。让我们从初始化函数开始：

```
void Init_Reset_Particles(void)
{
    //this function seves as both an init and reset for the particles

    // loop thru and reset all the particles to dead
    for(int index=0; index<MAX_PARTICLES;index++)
    {
        particles[index].state=PARTICLE_STATE_DEAD;
        particles[index].type=PARTICLE_TYPE_FADE;
        particles[index].x=0;
        particles[index].y=0;
        particles[index].xv=0;
        particles[index].yv=0;
        particles[index].start_color=0;
        particles[index].end_color=0;
        particles[index].curr_color=0;
        particles[index].counter=0;
        particles[index].max_count=0;
    } //end if
} //end init_Reset_Particles
```

Init_Reset_Particles() 仅仅给每个微粒赋予零值，准备使用它们。如果你想做任何特殊的事情，可以在这里做。我们所需要的下一个函数是用给定的初始条件启动一个微粒。我们会担心如何立刻达到初始条件，但是现在我要搜寻一个可行的微粒，如果找到的话就用传递的参数启动它。下面是这样做的函数：

```
Void Start_Particles(int type ,int color ,int countt,
Float x, float y,float xv,float yv)
{
    //this function starts a single particle

    int pindex=-1; // index of particle
    // first find open particle
    for(int index=0;index<MAX_PARTICLES;index++)
```



```

    if (particle[index.].state==PARTICLE_STATE_DEAD)
    {
        //set index
        pindex=index;
        break;
    } //end if
    //did we find one
    if(pindex!=-1)
        return;
    //set general state info
    particles[pindex].state=PARTICLE_STATE_ALIVE;
    particles[pindex].type=type;
    particles[pindex].x=x;
    particles[pindex].y=y;
    particles[pindex].xv=xv;
    particles[pindex].yv=yv;
    particles[pindex].counter=0
    particles[pindex].max_count=count;

    //set color ranges, always the same
    switch(color)
    {
        case PARTICLE_COLOR_RED:
        {
            particles[pindex].start_color=COLOR_RED_START;
            particles[pindex].end_color=COLOR_RED_END;
        }break;

        case PARTICLE_COLOR_GREEN:
        {
            particles[pindex].start_color=COLOR_GREEN_START;
            particles[pindex].end_color=COLOR_GREEN_END;
        }break;

        case PARTICLE_COLOR_BLUE:
        {
            particles[pindex].start_color=COLOR_BLUE_START;
            particles[pindex].end_color=COLOR_BLUE_END;
        }break;

        case PARTICLE_COLOR_WHITE:
        {
            particles[pindex].start_color=COLOR_WHITE_START;
            particles[pindex].end_color=COLOR_WHITE_END;
        }break;

    }break;

    } // end switch

    //what type of particle is being requested
    if (type==PARTICLE_TYPE_FLICKER)
    {

```

```

        //set current color
        particle[index].curr_color
        =RAND_RANGE(particles[index].start_color,
                    particles[index].end_color);

    }// end if
else
{
    //particle is fade type
    //set current color
    particles[index].curr_color=particles[index].start_color;
    }// end if

} //end Start_Particle

```

提示

这里没有错误检测或成功/错误的返回值。我并不关心这一点；如果我们不能创建一个细致的微粒，我想我一样能活下去。然而你可以增加更多的错误处理。

要用初速度 (0, -5) 在点 (10, 20) 启动一个微粒，要有90 帧的生命周期、退化的绿色，下面是你要做的：

```

Sstart_Particle(PARTICLE_TYPE_FADE,    //type
PARTICLE_COLOR_GREEN,    //color
90,                                //count,lifespan
10,20,                            //initial position
0,-5);                            //initial velocity

```

当然，点系统中既有引力又有风力，它们始终在起作用，所以可以在任何时候设定它们，这将会同时影响新微粒和已有微粒。如果你想没有风力，而仅有一点点引力的话，你就要这样做：

```

particle_gravity=0.1;    //positive is downward
particle_wind=0.0;    //could be +/-

```

现在我们必须决定怎样移动和控制这个微粒。我们确实想让它们环绕屏幕吗？或者当它们碰到边界时我们是不是应该消灭它？这取决于游戏的类型：2D、3D、滚动等等。现在让我们把问题简单化，当微粒跃出屏幕边界时统一使它自动消失。而且，移动函数应该修正颜色活度，检测生命数是否结束，消灭屏幕外的微粒。这里是移动函数，它把所有情况都考虑到了，包括引力和风力：

```

void Process_Particles(void)
{
    //this function moves and animates all particles

```

```

for(int index=0;index<MAX_PARTICLES;index++)
{
//test if this particle is alive
if(particles[index].state==PARTICLE_STATE_ALIVE)
{
//translate particle
particles[index].x+= particles[index].xv;
particles[index].y+= particles[index].yv;

//update velocity based on gravity and wind
particles[index].xv+=particle_wind;
particles[index].yv+=particle_gravity;

//now based on type of particle perform proper animation
if(particles[index].type==PARTICLE_TYPE_FLICKER)
{
//simple choose a color in the color range and
//assign it to the current color
particles[index].curr_color=
    RAND_RANGE(particles[index].start_color,
                particles[index].end_color);

//now update counter
if (++particles[index].counter>= particles[index].max_count)
{
//kill the particle
particles[index].state=PARTICLE_STATE_DEAD;
} //end if

} //end if
else
{
//must be a fade, be careful
//test if it's time to update color
if(++particles[index].counter>= particles[index].max_count)
{
//reset counter
particles[index].counter=0;

//update color
if(++particles[index].curr_color>
    particles[index].end_color)
{
//transition is complete, terminate particle
particles[index].state=PARTICLE_STATE_DEAD;
} //end if
} //end if
} //end else
}

```

```
//test if the particle is off the screen?
If (particles[index].x>screen_width||
    particles[index].y<0||
    particles[index].y>screen_height||
    particles[index].y<0)
{
    //kill it!
    particles[index].state=PARTICLE_STATE_DEAD;
} //end if

} // end if
} //end for index
} // end process_Particles
```

这个函数是自解释的——我希望。它转移微粒、应用外力、修正计数器和颜色、检测微粒是否离开屏幕，就是这样。下面我们需要绘制微粒。这可用许多种方法完成,但是我要假设简单的像素和一个后部缓冲显示，下面就是这样做的函数：

```
void Draw_Particles(void)
{
    // this function draws all the particles

    //lock back surface
    Ddraw_Lock_Back_Surface();

    For(int index=0;index<MAX_PARTICLES; index++)
    {
        //test if particle is alive
        if (particles[index].state==PARTICLE_STATE_ALIVE)
        {
            // render the particle,perform world to screen transform
            int x= particles[index].x;
            int y= particles[index].y;

            //test for clip
            if (x>screen_width||x<0||y>=screen_height||y<0)
                continue;
            //draw the pixel
            Draw_Pixel(x,y, particles[index].curr_color,
                Back_buffer,back_lpitch);
        } //end if
    } // end for index
    //unlock the secondary surface
    Ddraw_Unlock_Back_Surface();

} //end Draw_Particles
```

很兴奋吧？难道不想试试吗？我们几乎完成了。现在我们需要一些函数来创建像爆炸

和水气尾气的微粒影响。

产生初始条件

这部分十分有趣。你可以充分发挥你的想像力。让我们从一个水气尾迹算法开始。水气尾迹不过是从原始位置(emit_x,emit_y)释放出来的一些有不同生命周期和起始位置的微粒。

下面是一个可行的算法程序：

```
//emit a particle every with a change of 1 in 10
if((rand()%10==1)
{
    Start_Particle(PARTICLE_TYPE_FADE, //type
        PARTICLE_COLOR_GREEN, //color
        RAND_RANGE(90,150), //count,lifespan
        Emit_x+RAND_RANGE(-4,4), //initial x
        Emit_y+RAND_RANGE(-4,4), //initial y
        RAND_RANGE(-2,2), //initial x velocity
        RAND_RANGE(-2,2); //initial y velocity
} //end if
```

在发射器移动时，发射器的位置(emit_x,emit_y)也在移动，这样水气尾迹就留下了。如果你想得到逼真的效果，并且给水气微粒一个更真实的物理模型，你就应该仔细考虑发射器可以移动。这样任何被发射出的微粒都有最后的速度：发射速度+发射器速度。你需要知道发射源的速度（记作(emit_xv,emit_yv)），像下面这样简单的增加到微粒的最后速度中：

```
//emit a particle every with a change of 1 in 10
if((rand()%10==1)
{
    Start_Particle(PARTICLE_TYPE_FADE, //type
        PARTICLE_COLOR_GREEN, //color
        RAND_RANGE(90,150), //count,lifespan
        Emit_x+RAND_RANGE(-4,4), //initial x
        Emit_y+RAND_RANGE(-4,4), //initial y
        emit_xv+RAND_RANGE(-2,2), //initial x velocity
        emit_yv+ RAND_RANGE(-2,2); //initial y velocity
} //end if
```

让我们来模拟一个更叫人激动的爆炸。爆炸看起来就像是图13.43中所示一样。微粒在各个方向以球状被发射出来。

模拟起来够容易了。我们只需要从一个带有随机速度的普通点启动一个随机微粒数，它们在一个圆形半径上被平均分配。然后如果引力存在，微粒将落向地球或者由于生命周期的结束而消失在屏幕中。这里是爆炸程序的代码：

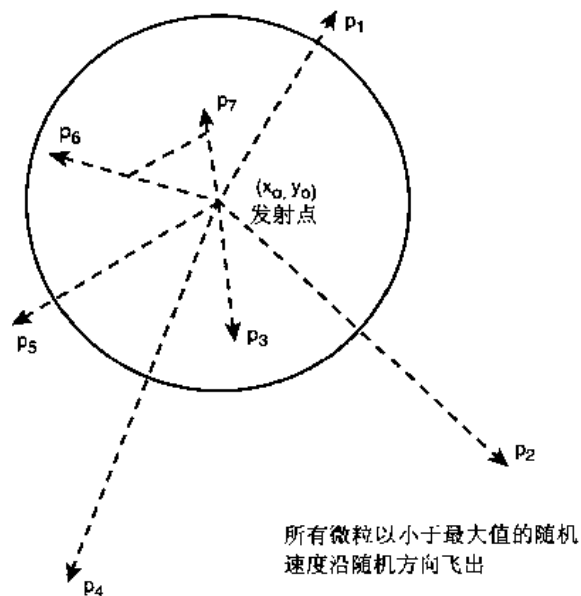


图 13.43 爆炸的微粒

```

void Start_Particle_Explosion(int type, int color, int count,
                             int x, int y, int xv, int yv,
                             int num_particles)
{
    // this function starts a particle explosion
    // at the given position and velocity
    //note the use of look up tables for sin,cos

    while (--num_particles>=0)
    {
        //compute random trajectory angle
        int ang =rand()%360;
        //compute random trajectory velocity
        float vel=2+rand()%4;

        Start_Particle(type,color,count,
                       X+RAND_RANGE(-4,4),y+RAND_RANGE(-4,4),
                       Xv+cos_look[ang]*vel,yv+sin_look[ang]*vel);
    } //end while
} //end Start_Particle_Explosion

```

`Start_Particle_Explosion()`采用具有你所期望的 (`PARTICLE_TYPE_FADE`, `PARTICLE_TYPE_FLICKER`)、颜色、数量及发射源的位置和速度的微粒类型。这个函数接着产生所有的期望的微粒。

要创造其他的特殊效果，只要写一个函数就行。例如，一个我喜欢在移动时采用的最酷的太空飞船驱散时的圆形冲击波。你只需要改变爆炸程序，用相同的速度启动所有的微

粒，但角度不同。下面是代码：

```
void Start_Particle_Explosion(int type, int color,int count,
                           int x,int y,int xv,int yv,
                           int num_particles)
{
    // this function starts a particle explosion
    // at the given position and velocity
    //note the use of look up tables for sin,cos

    //compute random velocity on outside of loop
    float vel=2+rand()%4
    while (--num_particles>=0)
    {
        //compute random trajectory angle
        int ang =rand()%360;

        //start the particle
        Start_Particle(type,color,count,
                      x,y,
                      xv+cos_look[ang]*vel,
                      yv+sin_look[ang]*vel);
    }
}
//end Start_Particle_Ring
```

集合微粒系统

现在你拥有了集合一些较酷的微粒效果所需要的所有东西。只要在你的游戏初始化阶段 `Init_Reset_Particles()` 中调用一下，然后在主循环中调用一下 `Process_Particles()` 就可以了。各个循环和引擎将完成剩下的任务。当然，你必须调用一些产生微粒的函数！最后，如果你想提高系统性能，可以增加内存管理以使你可以拥有无限的微粒，你也可以增加微粒与微粒、微粒与环境的碰撞探测——那将使效果更佳。

在用微粒系统的演示程序时，看一下CD中的程序 `DEMO13_10.CPP\EXE`。那是一个基于坦克工程的火炮演示程序。来自前面演示程序中的坦克发射现在的火炮。当然，记住我在演示程序中把微粒数增加到了256。

游戏关键：创建游戏的物理模型

这一章给出了许多信息和概念。关键是使用这些概念和一些重要数学知识来产生更好的模型。没有人知道，也没有人关心程序是否百分之百的模拟实际。如果你要估计那你就试试——只要你认为有价值。例如，如果你试图编制一个赛跑游戏，你想在公路上、冰上

或泥地中赛跑，你最好增加一些摩擦效果，否则你的汽车就像是在铁轨上一样。

换句话说，如果你有一个星空，玩游戏的人把它放大后每颗星裂成两颗或更多的小星星，那么我认为玩游戏的人并不关心或知道小星星将走的轨线——用确定的方式看起来同样逼真。

物理模拟的数据结构

一个经常提出的问题（怎样用VC++来解释DirectX程序）就是物理模拟用的数据结构。可是并没有物理数据结构！大多数物理模型是基于游戏对象本身——你只需要给初始数据结构增加足够的数据单元来解释物理模型——可以了吗？虽然如此，你应该清楚下面的在任何物理引擎中为宇宙和物体而设的参数和值：

- 物体位置和速度。
- 物体的角速度。
- 物体的质量、摩擦系数及任何其他的物理特性。
- 物体的物理几何形状。这里用的是可用于物理计算的简单几何学。你可以采用矩形、球形或其他，但不用实际的物体几何形状。
- 外力，如风、引力等等。

现在，轮到你来用适当的结构或类型来描述这些值了。例如，实际碰撞演示程序采用这样的模型：

```
float x,y;      //position
float xv,yv;    //velocity
float radius;   //guess?
Float coefficient_of_restitution; //just what it says
```

当然数据被隐藏在描述每个球的BOB内部数组中，但抽象的数据结构正是我们感兴趣的。

基于帧与时间的模拟

这是我要谈论的最后一个主题，因为它在3D游戏中越来越重要。这样在本书的很大篇幅，我们要讲述像图13.44那样的游戏循环。我们已经假定游戏是以恒定远距离传输率运行的。如果不是这样，就不用作较大处理，所有的东西都将很慢的下载到屏幕上。但是如果你不想让下载速度很慢，你不管传输率是多少而要让一艘船从a到b只用两秒钟，这就是基于时间的模拟。

基于时间的模拟不同于基于帧的模拟之处在于时间被用于所有的移动物体动力学方程上。例如，基于帧的游戏中如果你有这样的代码：

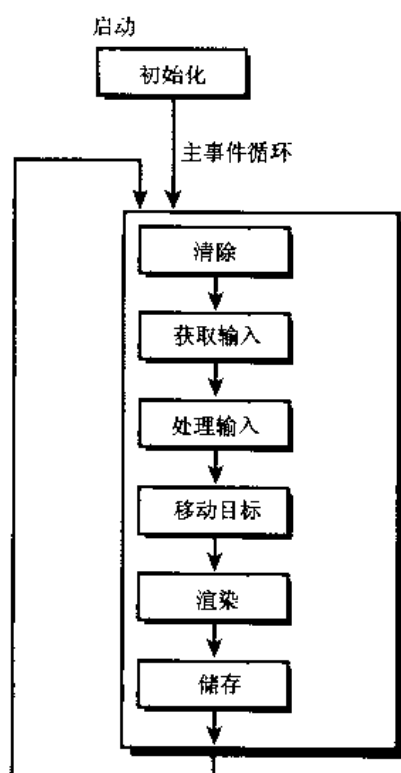


图 13.44 游戏循环

$x=0, y=0$

$x=x+dx$

$y=y+dy$

游戏以30fps的速率运行，那么在30帧或1秒之后， x 、 y 将为：

$x=30 \times dx$

$y=30 \times dy$

如果 $dx=1, dy=0$ ，那么物体将在 x 方向上恰好运行30个像素。如果你可以总是精确地保证恒定传输率的话当然很好了。但是传输率降到10fps会怎样呢？那么过1秒后：

$x=10 \times dx=10$

$y=10 \times dy=0$

x 仅仅变化了你需要的三分之一。如果你侧重于视觉连贯性，那么这是不可接受的。而且，这可能严重破坏一个网络游戏。在网络游戏中你可以与最慢的机器保持同步而保持帧，或者你也可以让所有的机器自由运行而使用时间模拟，这样会更现实且容易作到。我当然没有理由在我有PIII-1000MHz的机器时却为其他的486机主付钱。

要执行基于时间的运动和运动学，你必须在你的所有行动方程中使用时间。然后，当到了移动物体的时间时，你必须从上一个运动开始检测时间的差别，且把它作为你的方程的输入。

这样，虽然游戏慢了好多，但却不碍事，因为时间参数将控制进度而且会带来较大的移动。这里是游戏中循环的一个例子。

```
While(1)
{
    t0=Get_Time(); //assume this is in milliseconds

    //work,work,work

    //move objects
    t1=Get_Time();

    //move all the objects
    Move_Objects(t1-t0);

    //render
    Render();

} // end while
```

用这个循环我们使用了时间变换或 $(t_1 - t_0)$ 作为输入给移动代码。通常没有输入；移动代码也只是“移动而已”。让我们假设我们的物体以每秒30像素的速度移动，但因为我们的时间基础是微秒或 1×10^{-3} 秒，我们这样做：

$dx = 30 \text{ 像素/秒} = 0.03 \text{ 像素/微秒}$

你能看出来我打算怎么做吗？让我们为 x 写出运动方程：

$$x = x + dx \times t$$

插入每一件事情我们都可以得到

$$x \approx x + 0.03 \times (t_1 - t_0)$$

就是这样。如果一个单独的帧用了1秒，那么 $(t_1 - t_0)$ 将是1000微秒，移动方程将是

$$x \approx x + 0.03 \times 3 = 0.09, \text{ 也是正确的!}$$

很显然你需要使用浮点数来工作，因为你打算追踪像素移动的片段，但你知道怎么做。很不错吧，因为即使你的程序由于表演而使核心部分开始陷入困境，移动也将同样进行。

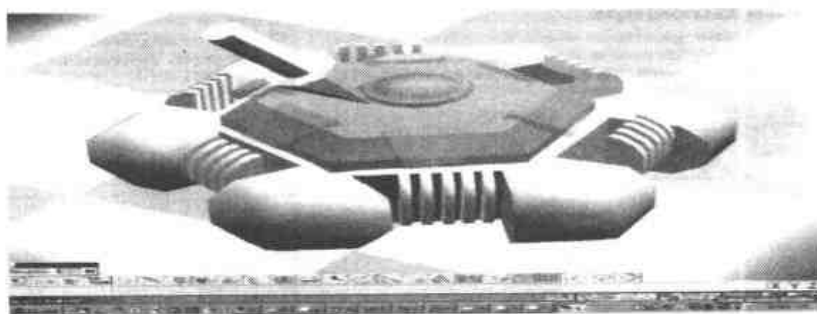
参考程序DEMO13.CPPIEXE;它主要从左向右移动一只小船（带有影子），允许你使用方向键改变每一帧的延迟从而控制过程加载。

注意船是以恒定速率移动的。它可以跳，但不管帧的传输率是多少，它总是以50像素/秒的速度进行。作为一个测试，屏幕是640像素宽，加上屏外交迭的160像素，这样总数为800像素。因为船的行进速度是50像素/秒，也就是说走完要用 $800/50=16$ 秒。试着改变一下延迟，同时记住这总是对的。如果你的游戏设计成60fps的传输率，而降到15~30后跳动将不会那么明显，游戏看不出什么不同，但不太流畅。如果没有时间模拟，你的游戏将运行得很慢，就像在雪地中行进一样——我相信你已经看到这种情况不止一次了。

总 结

我相信这一章对于任何人都是有启发性的，不管你的水平如何。写这一章是很叫人头疼的。我们接触到许多方面，了解了各种看待问题的方式。例如，我们特别尝试一个用于处理球弹出矩形的碰撞反应运算算法。这个技术适用于任何类型的跳跃游戏。然后我们仔细地学习了相关地数学知识，并且用通常的正确方法来实现。这是游戏中物理模拟的关键点——你只要学会怎样做就行了。

14



综 合 运 用

这一章我打算略述一个简单游戏——*Outpost* 的设计和执行，这个游戏用到的技术都是本书中曾学习过的知识。

这个游戏用了五天时间就完成了，所以别对它期望太高。然而，它游戏的内容有 3D 模型、微粒、声效和不同的“敌人”，我认为对你来说使用它是很容易的。下面是我要介绍的内容：

- ➔ *Outpost* 的初步设计
- ➔ 用于编程的工具
- ➔ 游戏领域：太空滚动
- ➔ 游戏者的飞船：鬼怪号
- ➔ 行星领域
- ➔ 敌人
- ➔ 能量供给
- ➔ HUD
- ➔ 微粒系统
- ➔ 执行游戏
- ➔ 编译 *Outpost*

Outpost 的初步设计

我想创建一个容易编制、界面友好、有扩展性及使用滚动的游戏程序。这样，我挑选

了一个星类空间游戏，因为它所需惟一的背景是黑色太空。给行星加上人工智能，敌人加上搜索与毁坏智能，这些都是很简单的，所以这个想法真是个好主意。

故事概要

故事的情况大概是这样的：你是极机密的鬼怪号飞船的飞行员，一个被送到 11 号区域去执行抵御外国侵略任务的全副武装的斗士。侵略者已经在这个区域大批出现，你必须消灭他们。存在的问题是这个区域充满着（行星的）残骸、入侵者的飞船和保护每个边哨地雷。这就是故事的大概情况（听起来就像是一部电影）。图 14.1 和图 14.2 是游戏中的启动和执行时的画面。

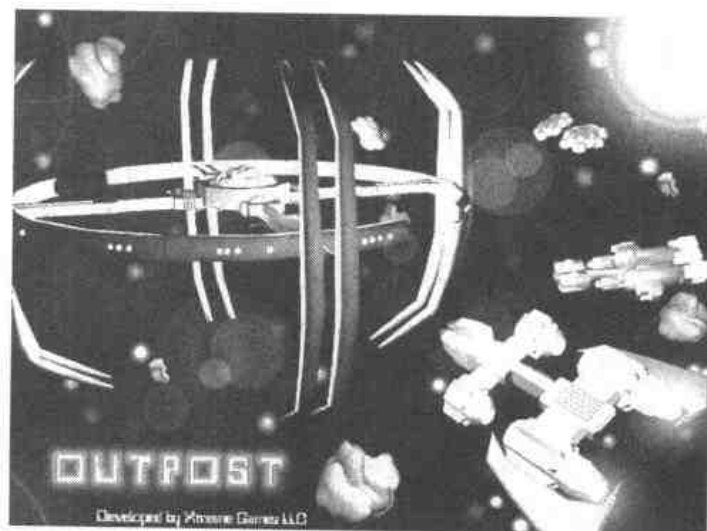


图 14.1 启动 Outpost

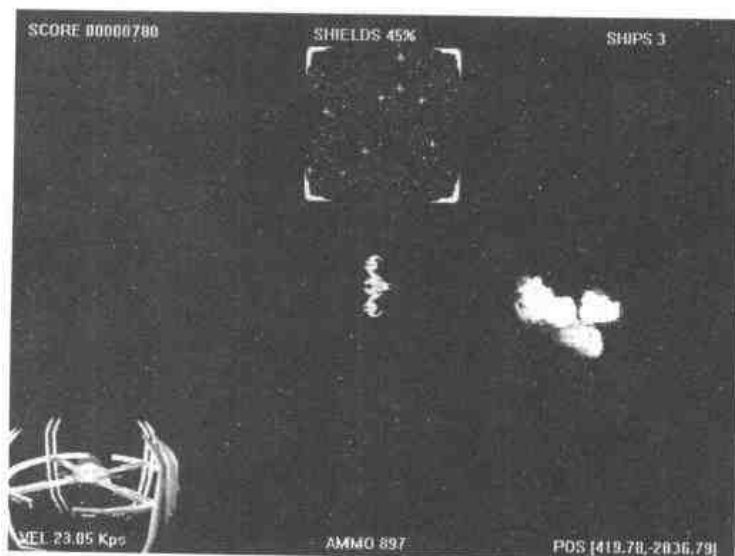


图 14.2 Outpost 在执行中

通常我认为构思游戏情节是创建游戏程序中最重要的一环，这是你的工作重点和保持游戏连贯性的关键。

设计游戏程序

完成了故事情节的设计，我就开始考虑游戏程序的编制。这一点取决于编制的规则，什么对象或控制完成什么功能，目的是什么等等。游戏并不复杂，所以我主要考虑游戏者能做什么，敌人能做什么，游戏者怎样胜利或失败，每一样事物的人工智能等等。因为 Outpost 并没有太高的水准，也没有太多的策略，所以没有多少东西要设计。

编制游戏的工具

因为我在书中用了 256 色模式使问题简化，所以这里仍然选用 256 色，但我还很想清晰地看到物体，因此我几乎要渲染每个物体。我使用 TS4，这可能是考虑价格后最好的制作 3D 模型的工具。然而，不考虑价格，TS4 能很好地运行 3D Studio Max。CD 上有一个 TS4 的演示拷贝，所以记住要检测一下。

2D 艺术画面的创建和渲染使用 JASC Paint Shop Pro5.1 完成，这是我看到过的最方便的界面及支持绘画包的工具。同样，CD 上也有一个演示的拷贝。

最后，声音效果从各种声源中得到，使用 Sonci Foundry 的 Sound Forge XP 完成，这是应用于 PC 机上最好的声音编辑软件包了。你也可以在 CD 上找到一个 Sound Forge 的演示拷贝。这些就是我要使用的工具，此外还有 VC++5.0、6.0 和 DirectX。

表 14.1 列出了游戏中所有的对象及创建方法。

表 14.1 Outpost 中的对象

物 体	技 术
行星	TS4
飞船	用 PSP 手工画出
边哨	TS4
掠夺者的地雷	TS4
武装直升机	TS4
供给能量	TS4
等离子脉冲	用 PSP 手工画出
星空	单个像素
爆炸	数字化的 Stock Media

所有 3D 模型都是用 TS4 手工画出来的，每个都用了不到一个小时。每幅手工图像则大约用了一个小时。旋转游戏者的飞船是在 PSP 中用旋转算法实现，而不是手动完成。所有的声音都是以 11kHz、8 位采样。

游戏领域：空间滚动

我已经介绍过如何执行滚动，所以这里不再介绍了。然而，Outpost 中有两个有趣的滚动细节。第一，游戏者始终在屏幕的中间。这使问题变得简单了，但更重要的是给游戏者提供了最大的行动空间。如果你允许游戏者可以到达屏幕的边缘，那么在滚动游戏中敌人可能在你行动做出反应的几毫秒时间内出现和消灭你。通过让游戏者始终处于中间就给他一个开阔的视野，让他看清楚自己周围的一切。

至于游戏空间的大小我选择了 16000×16000 ，如下：

```
// size of universe
#define UNIVERSE_MIN_X    (-8000)
#define UNIVERSE_MAX_X    (8000)
#define UNIVERSE_MIN_Y    (-8000)
#define UNIVERSE_MAX_Y    (8000)
```

空间尺寸总是很难确定，但我的标准技术是这样的：估计一下游戏运行时每秒有多少画面，估计游戏者移动的最大速度，然后确定一下游戏者从一端到另一端的大概时间：

```
universe_size=player_velocity*fps*desired_time
```

因此，如果游戏者每秒中最多移动 32 像素/画面，你想在 10 秒内以 30fps 的速度从一端移到另一端，那么空间尺寸就是：

```
Universe_size=32pixcls/frame*30fps*10
             -9600units
```

这里每个单元是一个像素。这就是我怎样得出 16000 的。当然游戏中最后的值有一些不同，但这是我得出棒球场尺寸的方法。

在你建立滚动空间游戏时要考虑的其他问题就是稀疏性。你可能认为 10×10 秒或 16000×16000 像素的空间不够大，但即便这样要充满整个空间也需要上百个行星。否则游戏者将到处飞行去寻找目标射击！我的朋友 Jarrod Davis 在写 Astro3D 时发现了这一点，你可以在 CD 上找到。

至于滚动算法，并没有太多的东西要说。图 14.3 解释了游戏者的位置始终处于中间的原理。

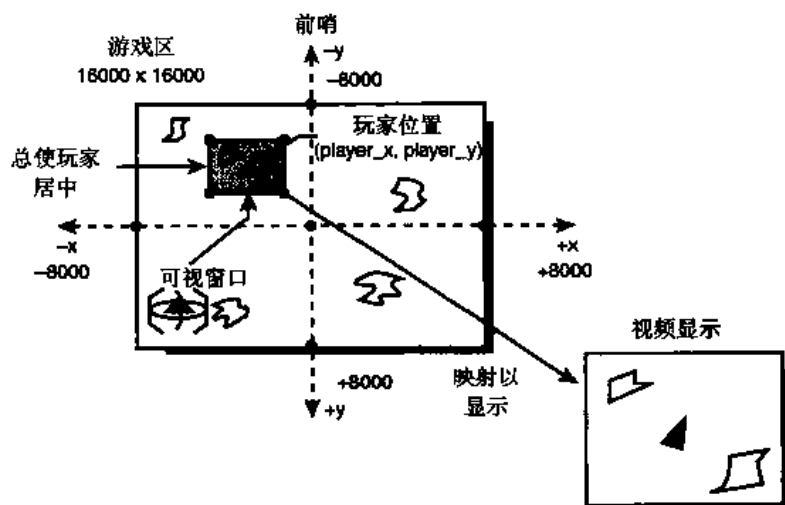


图 14.3 用于 Outpost 的滚动算法

算法通过找到游戏者的工作位置，调节游戏者始终处于原始位置（屏幕中间），调动物体朝游戏者飞去。窗口内的物体被渲染过而其他的则没有。惟一没有渲染过的物体就是星星，它们是最基本的像素，整群的移出屏幕。因此如果你在 x 或 y 轴方向移动速度很慢的话，你可以在同一位置看到同样的星星。

游戏者的飞船：“鬼怪号”

游戏者的飞船是手工画出来的。我要对它进行渲染，但它的细节太多，因此我决定画出来后使用 Paint Shop Pro 来使它发光，从而使之看起来像真的一样。除此之外，我只画出了“鬼怪号”朝南飞行的图像，然后使用 PSP 来处理得到飞船旋转的十六个角度。这件艺术品如图 14.4 所示。

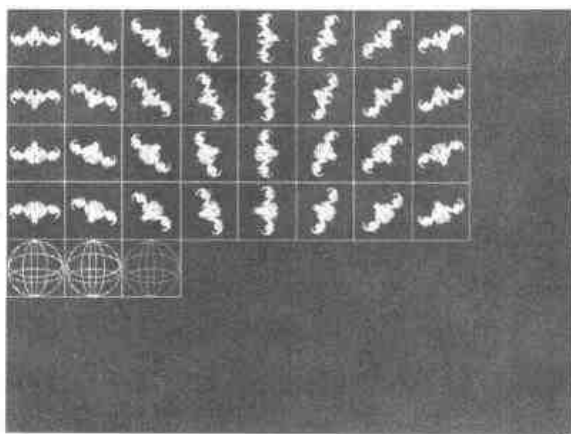


图 14.4 鬼怪号的艺术设计

鬼怪号除了飞行和射击之外没有其他太多的功能，但摩擦算法却很有趣。我想让鬼怪号看起来像在引力下飞行，所以我想给它一些转弯时的摩擦形式。我用以前章节中介绍过的技术解决了这个问题。

游戏者以任意方向推进时鬼怪号的速度矢量都随之更动。这种解决途径的好处在于你不用担心这样的问题：“如果飞船正在朝东飞，游戏者又向南推进，飞船该怎么办？”可以用数学矢量来做。

下面是对鬼怪号的控制代码：

```
// test if player is moving
if (keyboard_state[DIK_RIGHT])
{
    // rotate player to right
    if (++wraith.varsI[WRAITH_INDEX_DIR] > 15)
        wraith.varsI[WRAITH_INDEX_DIR] = 0;

    } // end if
else
if (keyboard_state[DIK_LEFT])
{
    // rotate player to left
    if (wraith.varsI[WRAITH_INDEX_DIR] < 0)
        wraith.varsI[WRAITH_INDEX_DIR] = 15;

    } // end if

// vertical/speed motion
if (keyboard_state[DIK_UP])
{
    // move player forward
    xv = cos_look16[wraith.varsI[WRAITH_INDEX_DIR]];
    yv = sin_look16[wraith.varsI[WRAITH_INDEX_DIR]];

    // test to turn on engines
    if (!engines_on)
        DSound_Play(engines_id,DSBPLAY_LOOPING);

    // set engines to on
    engines_on = 1;

    Start_Particle(PARTICLE_TYPE_FADE, PARTICLE_COLOR_GREEN, 3,
        player_x+RAND_RANGE(-2,2),
        player_y+RAND_RANGE(-2,2),
        (-int(player_xv)>>3), (-int(player_yv)>>3));

    } // end if
else
if (engines_on)
{
    // reset the engine on flag and turn off sound
    engines_on = 0;
```

```
// turn off the sound

DSound_Stop_Sound(engines_id);
} // end if

// add velocity change to player velocity
player_xv+=xv;
player_yv+=yv;

// test for maximum velocity
vel = Fast_Distance_2D(player_xv, player_yv);

if (vel >= MAX_PLAYER_SPEED)
{
    // recompute velocity vector by normalizing then rescaling
    player_xv = (MAX_PLAYER_SPEED-1)*player_xv/vel;
    player_yv = (MAX_PLAYER_SPEED-1)*player_yv/vel;
} // end if

// move player; note that these are in world coords
player_x+=player_xv;
player_y+=player_yv;
```

提示



关于鬼怪号的名字，很久以前我看过一部叫“The Wraith”的 Charlie sheen 主演的电影。在这部电影中，Charlie 在他的车厢中结束了生命，因为车子以 32 英尺/秒的速度跌落到悬崖下。他死后作为鬼魂又回来，目的是要杀掉谋杀者报仇。无论如何，他有很酷的车子，原始隐形拦截机，我就是从这部电影得到这个名字的。

研究一下这些代码，你会注意到有一些语句的作用是让鬼怪号不要走得太快。我继续检测速度矢量的长度，使之不超过最大值。如果太长我就会缩短它。另一种方法就是使用单元方向矢量和速度标量，以单元方向矢量和速度标量调动飞船。两种方法效果一样。

最后，鬼怪号有一个防护罩和水气尾迹。防护罩不过是鬼怪号撞击时我覆盖上的一个位图，水气尾迹就是推进器工作时随机释放的粒子。推进效果通过在每个方向上的两个位图取得：一个是推进器启动和一个推进器脱离。当推进器启动时我随机选出启动图像，就像是 Klingon 脉冲驱动一样。

行星领域

行星领域由许多随机飞行的星状物实现，共三种尺寸：小、中和大。行星用 TS4 实时渲染。图 14.5 是模拟中的行星，图 14.6 是渲染过的情况。除了使行星看起来很逼真并且正确地发光，我还创建了一个旋转的效果，每个行星都有 Targa(TGA)的旋转移动功能。把 TGA 文件 image0000.tga 和 image0001.tga 转化为位图，然后不用模板直接输入到游戏中。

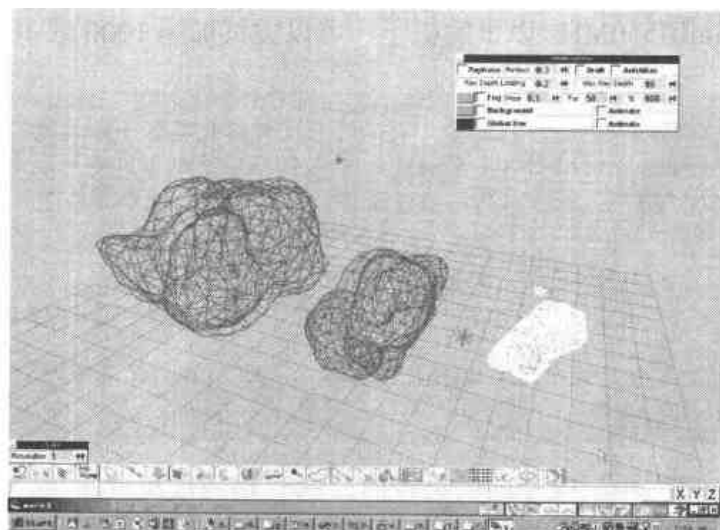


图 14.5 行星的 3D 模型

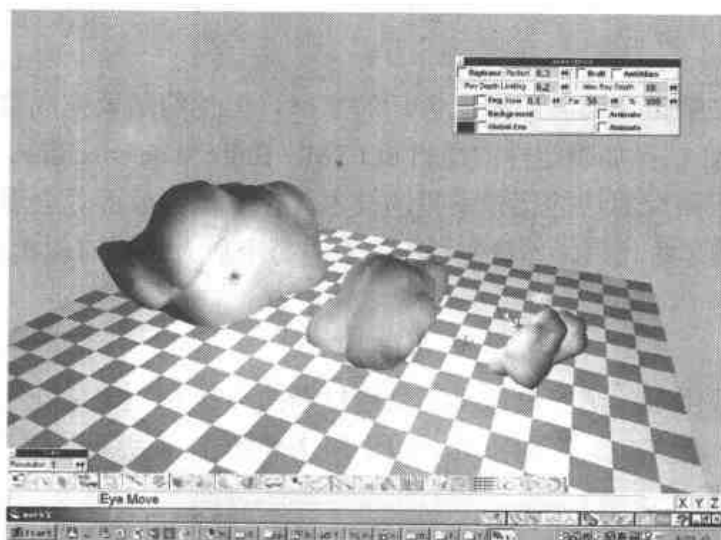


图 14.6 渲染过的行星

行星的物理模型很简单。它们只是以不同的速度朝同一个方向移动，直到它们被击中为止。除此之外行星以变化的速度旋转从而给人整体的感觉。当行星被击中时它的尺寸和硬度决定它怎样爆炸。你可以用一个冲击波击中大行星以使它完全消失，但有时却是裂开。裂开决定于两个因素：可能性和有效性。中行星和小行星数量有一定，所以不能总是裂成两颗。

此外，我并不喜欢大行星裂为两个中的，或一颗中行星裂为两块小行星的做法。所以，有时候一块大的会裂成一块中等大小和两块小行星。为使游戏更有趣，在边上发射一些可能的变化的行星。如果你想增加更多的行星，改变下面的定义：

```
#define MAX_ROCKS      300
```

如果你有 PentiumIII550MHz 以上的机器，可以尝试加到 1000 或 10000。

技巧



我对游戏中的行星很喜爱。大学时一次打赌我赢了 100 美元。一些其他计算机系的学生赌我不能在他们面前用 Pascal 语言在 IBM XT 机上正确地写出行星游戏。他们看过我写的其他游戏，说我是拷贝的。当然，他们就是那些典型的除非有 API 可调用否则什么都不能做的计算机科学的学生。

我坐下来花了八小时写出行星游戏——一个确实的 Atari 的矢量版本的拷贝（尽管没有声音）。我赢得 100 美元，然后我用手背拍了下他们，取走了他们口袋里的钱。

关键是我在做了很多次后记得行星的代码，我总是喜欢把它作为一个例子，就像我的“hello world”一样。

最后，当一个行星击中游戏空间的边缘时，它只是通过重置 x 或 y 位置变量而隐藏起来。但你是否想让它通过速度矢量的反射而从边界弹起来呢？

敌人

游戏中的敌人在宇宙空间并不是最灵活的家伙，但他们却要完成自己的任务。他们大部分使用介绍过的 AI 法，如确定性的逻辑和 FSM（finite state machines，有限状态机制），然而，还有一种用于跟踪算法的比较不错的技术。在本章的后面你会看到使入侵者在游戏者的位置埋下地雷的那些代码。不管怎样，先看看每个敌人是如何创建及执行的。

边哨

在整个游戏中我用于边哨的模型可能是最复杂的 3D 模型。它花了我一个小时。最叫人头疼的是这个 3D 模型有太多的细节，如图 14.7 和图 14.8 所示，但是我对它渲染和压缩时忽略了所有的细节。

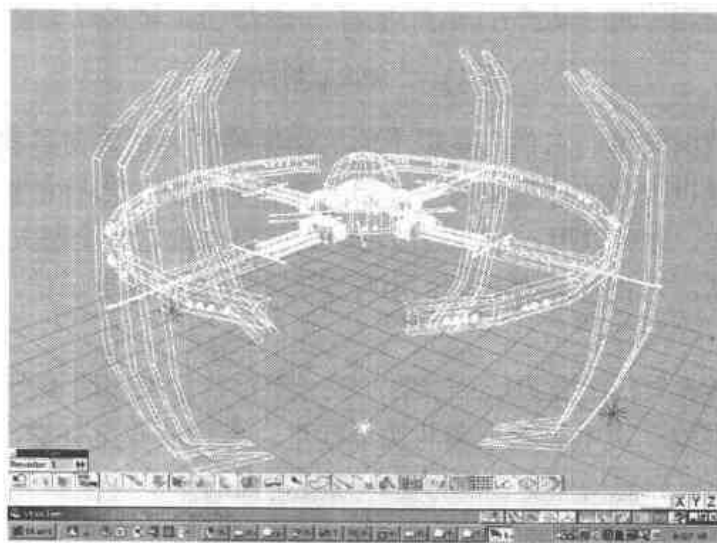


图 14.7 边哨的 3D 模型

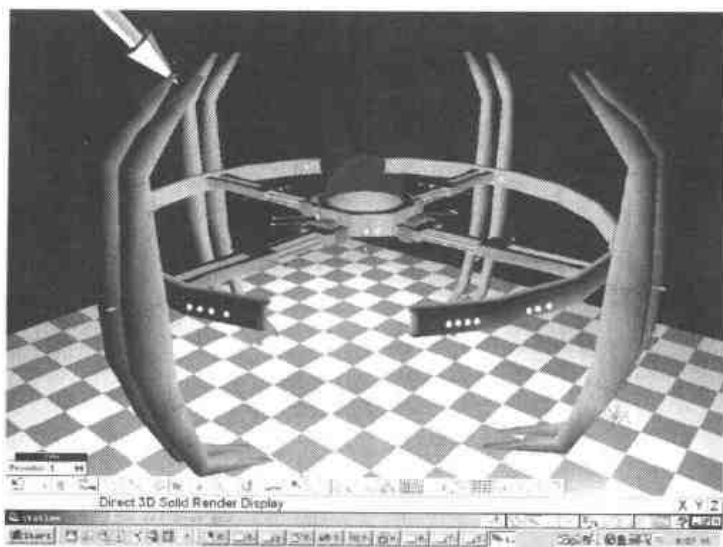


图 14.8 渲染过的边哨

地雷是边哨的保护者。不管怎样，边哨除了停在那里和旋转外没有太多其他的功能。它们没有武器，没有 AI，什么都没有。然而它们可以探测毁坏，当游戏者击中它们时它们就开始爆炸。边哨的毁坏程度非常大，以至于发生爆炸时粒子和二次爆炸都会产生！

入侵者的地雷

地雷是边哨的保护者。它们在附近占据着位置，直到你在一定的特别范围内出现，然后它们就转向和跟踪你。它们用 TS4 渲染，如图 14.9 和图 14.10 所示。

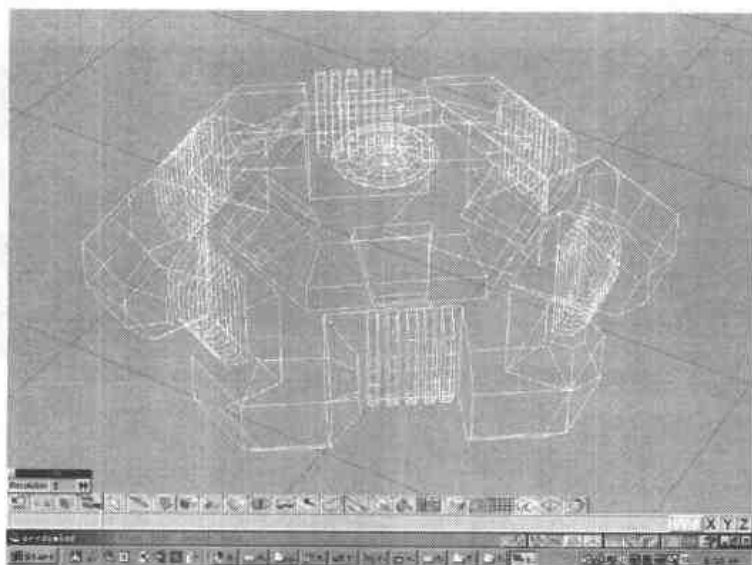


图 14.9 入侵者的地雷的 3D 模型

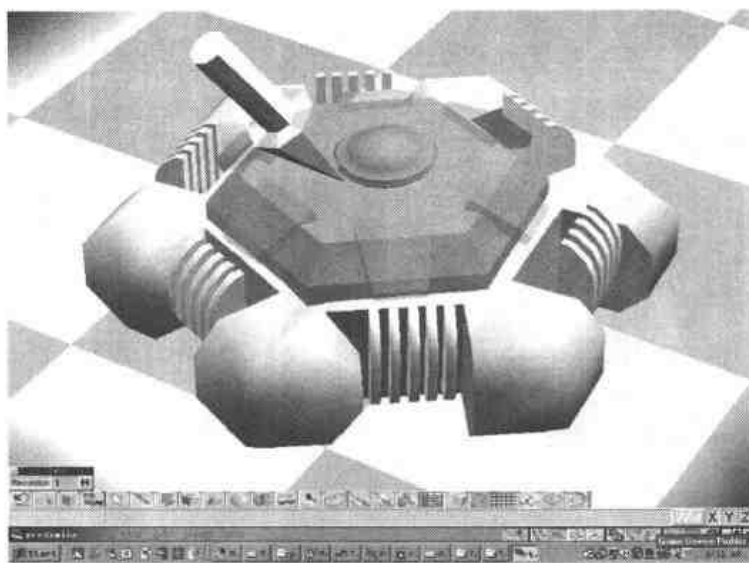


图 14.10 渲染过的入侵者地雷

对于最后为它们使用的 3D 模型我并不满意。实际上我创建了另一个 3D 模型,如图 14.11 所示,但它看起来更像一个固定的雷,而不是可以向你进攻的武器。

在任何情况下,用于入侵者的地雷的 AI 都很简单。它是从空转或睡眠状态启动的有限状态的机器。当你到了一定的范围内它就会活动,它是以下面的定义为基础的:

```
#define MIN_MINE_ACTIVATION_DIST 250
```

如果游戏者的位置在入侵者地雷的有效范围内,可以用十二章“人工智能在游戏中的运用”中介绍的人工智能算法激活它们。

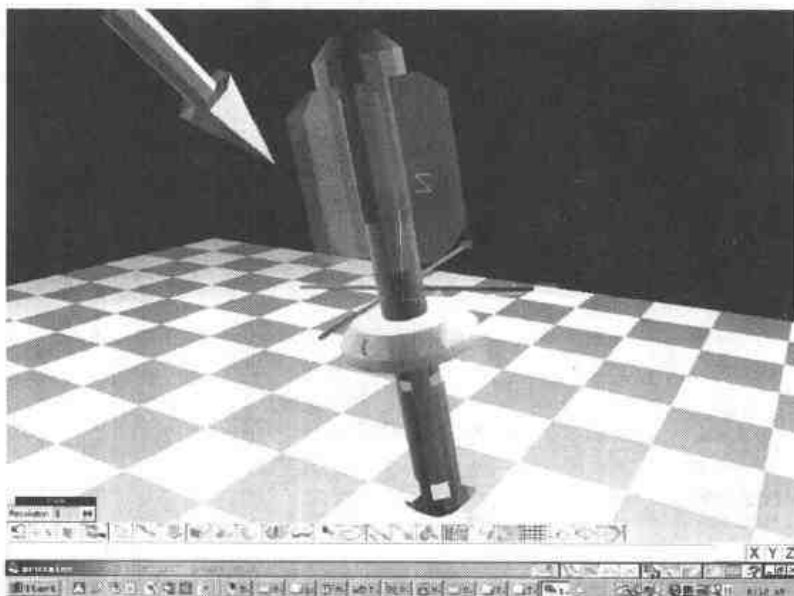


图 14.11 另一个入侵者地雷的概念

入侵者的地雷并没有武器，它们只是靠近你，对你的飞船进行破坏。

武装直升机

武装直升机是用 TS4 制作出来的，如图 14.12 和图 14.13 所示。它们的细节太多，并且看起来非常庞大，我只得对其进行压缩并用 256 色覆盖。

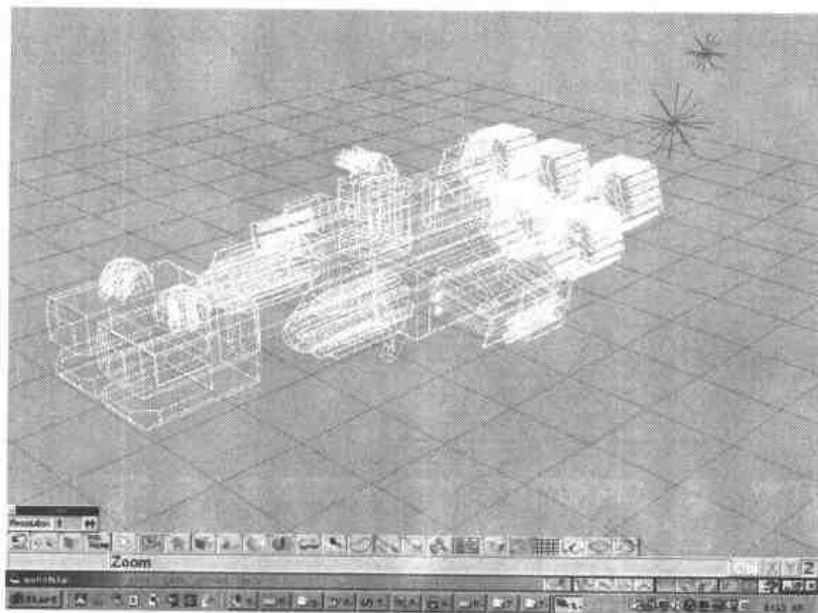


图 14.12 武装直升机的 3D 模型

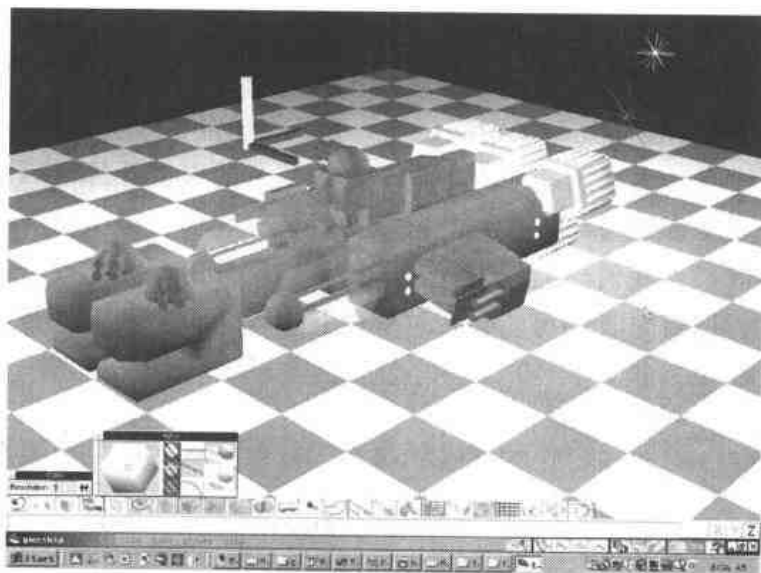


图 14.13 渲染过的武装直升机

武装直升机的 AI 也很简单。它们以恒定的速度在 x 轴方向移动。如果它们与游戏者的距离在一个特定的范围内，就会调整 y 轴方向的位置来跟踪游戏者，但效率极低。因此，

游戏者总是可以迅速地直接改变方向逃跑。武装直升机的威力在于它的重武器。每架武装直升机都装备了三门可以单独开火的激光炮，如图 14.14 所示。

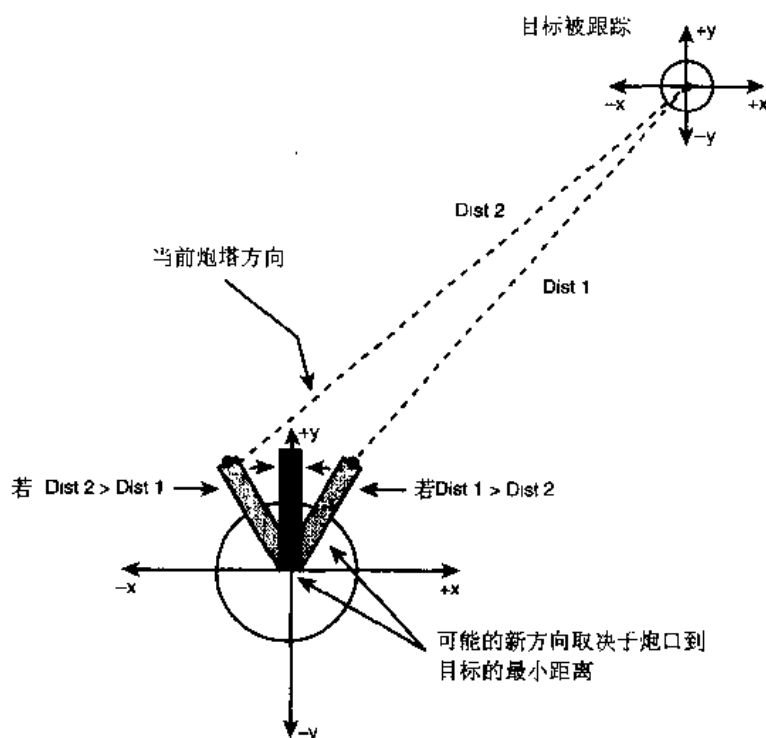


图 14.14 武装直升机的炮塔瞄准算法

大炮的瞄准算法是很酷的。它通过设计炮塔指向的方向矢量和另一个从炮塔到游戏者飞船的矢量来工作。然后程序找出炮塔头部到飞船的最短距离，所以它顺时针和逆时针都检测一次，以便得出哪个旋转是最短距离，然后完成得出最小距离的旋转。

这个算法程序没有什么潇洒的地方，也没有很复杂的矢量计算，只是使用了距离计算和最小距离算法。我是仔细考虑了人们的脑袋如何追踪物体后得出这样的算法的。我们以物体的方向开始转向，当我们感到我们以正确的方向动作时，我们就开始减慢我们脑袋的转动，然后停下来。但有时我们可能会转过头并且不得不调整。这是算法的灵感。看一下下面的代码：

```
// first create a vector point in the direction of the turret

// compute current turret vector
int tdir1 = gunships[index].varsI[INDEX_GUNSHIP_TURRET];

float dlx = gunships[index].varsI[INDEX_WORLD_X] +
            cos_look16[tdir1]*32;
float dly = gunships[index].varsI[INDEX_WORLD_Y] +
            sin_look16[tdir1]*32;
```



```

// compute turret vector plus one
int tdir2 = gunships[index].varsI[INDEX_GUNSHIP_TURRET]+1;

if (tdir2 > 15)
    tdir2 = 0;

float d2x = gunships[index].varsI[INDEX_WORLD_X] +
    cos_look16[tdir2]*32;
float d2y = gunships[index].varsI[INDEX_WORLD_Y] +
    sin_look16[tdir2]*32;

// compute turret vector minus one
int tdir0 = gunships[index].varsI[INDEX_GUNSHIP_TURRET]-1;

if (tdir0 < 0)
    tdir0=15;

float d0x = gunships[index].varsI[INDEX_WORLD_X] +
    cos_look16[tdir0]*32;
float d0y = gunships[index].varsI[INDEX_WORLD_Y] +
    sin_look16[tdir0]*32;

// now find the min dist
float dist0 = Fast_Distance_2D(player_x - d0x,
    player_y - d0y);
float dist1 = Fast_Distance_2D(player_x - d1x,
    player_y - d1y);
float dist2 = Fast_Distance_2D(player_x - d2x,
    player_y - d2y);

if (dist0 < dist2 && dist0 < dist1)
{
    // the negative direction is best
    gunships[index].varsI[INDEX_GUNSHIP_TURRET] = tdir0;

} // end if
else
if (dist2 < dist0 && dist2 < dist1)
{
    // the positive direction is best
    gunships[index].varsI[INDEX_GUNSHIP_TURRET] = tdir2;
} // end if

```

技巧



你会注意到我使用了许多距离计算。然而，它们都基于函数 `Fast_Distance()`，所以速度很快，时间不会超过几次移位和加法的时间。

供给能量

我觉得使用无限的弹药和防护罩会使游戏感觉有些缺乏战术性，因此我考虑这样一个问题“为什么不搞供给能量？”脑袋中有了那个想法后，我就坐下来开始用 TS4 模拟供给能量。之后我认识到我并没有太好的模拟观点。

供给能量通常并不现实。我的意思是，它们的表面写着“AMMO”，在反引力的驱动下在空间飘动等，但仍然要正确的看待它们。最后我决定用写着“AMMO”和“SHLD”的透明球体。如图 14.15 所示是 3D 模型。

一旦作好了 3D 模型，我就会对它们进行渲染，准备用在游戏中。我想让它们在一个行星或敌人被击中时出现，最后我决定它把它放在行星中更有意义。我的结论就是当你毁掉一个行星、残骸和珍贵的矿物（如水晶）时在爆炸期间都可出现。听起来很合理。

当一个行星毁掉后，一个好像十面体的骰子就滚出来了。如果骰子落地时 TRUE 的一面朝上，一个供给能量（弹药或防护罩）就产生了，它以很低的速度从爆炸处飞出来。当你撞上它后你就会吸收它的材料，或你的防护罩和弹药会增加。

一开始，我创建了供给能量并且让它飘起来。但很快我发现空间过于巨大，第二我发现看不见它们的身影了，确实很难发现。我可以把雷达点放在扫描器上，但使供给能量有一个生命期更有意义。因此，当供给能量产生后，它存活 3~9 秒后就会消失。这样就可有无限数量的供给能量，如果你丢失了一个，也可获得再生，不仅仅只在空间消失。



图 14.15 渲染过的供给能量

HUD

用于 Outpost 的 HUD(Heads-up display, 状况显示)由两个主要的组件组成: 扫描器和一些战术信息, 包括燃料、速度、防护罩、弹药、飞船数量和级别。如图 14.16 所示。它们都用漂亮的外部绿色装扮过了——我喜欢绿色。战术信息是简单的 GDI-文本, 但扫描器是用线、位图和像素的 DirectX 工具来制作。

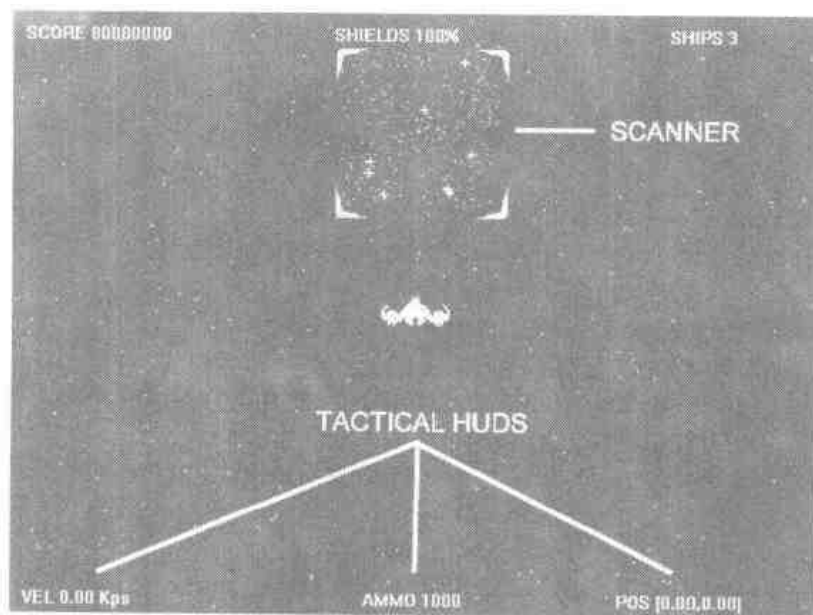


图 14.16 HUD

看一下扫描器的代码:

```
void Draw_Scanner(void)
{
    // this function draws the scanner

    int index,sx,sy; // looping and position

    // lock back surface
    DDraw_Lock_Back_Surface();

    // draw all the rocks
    for (index=0; index < MAX_ROCKS; index++)
    {
        // draw rock blips
        if (rocks[index].state==ROCK_STATE_ON)
        {
```

```

        sx = ((rocks[index].varsI[INDEX_WORLD_X] -
            UNIVERSE_MIN_X) >> 7) +
            (SCREEN_WIDTH/2) -
            ((UNIVERSE_MAX_X - UNIVERSE_MIN_X) >> 8);
        sy = ((rocks[index].varsI[INDEX_WORLD_Y] -
            UNIVERSE_MIN_Y) >> 7) + 32;

        Draw_Pixel(sx,sy,8,back_buffer, back_lpitch);
    } // end if

} // end for index

// draw all the gunships
for (index=0; index < MAX_GUNSHIPS; index++)
{
    // draw gunship blips
    if (gunships[index].state==GUNSHIP_STATE_ALIVE)
    {
        sx = ((gunships[index].varsI[INDEX_WORLD_X] -
            UNIVERSE_MIN_X) >> 7) +
            (SCREEN_WIDTH/2) - ((UNIVERSE_MAX_X -
            UNIVERSE_MIN_X) >> 8);
        sy = ((gunships[index].varsI[INDEX_WORLD_Y] -
            UNIVERSE_MIN_Y) >> 7) + 32;

        Draw_Pixel(sx,sy,14,back_buffer, back_lpitch);
        Draw_Pixel(sx+1,sy,14,back_buffer, back_lpitch);

    } // end if

} // end for index

// draw all the mines
for (index=0; index < MAX_MINES; index++)
{
    // draw gunship blips
    if (mines[index].state==MINE_STATE_ALIVE)
    {
        sx = ((mines[index].varsI[INDEX_WORLD_X] -
            UNIVERSE_MIN_X) >> 7) +
            (SCREEN_WIDTH/2) - ((UNIVERSE_MAX_X -
            UNIVERSE_MIN_X) >> 8);
        sy = ((mines[index].varsI[INDEX_WORLD_Y] -
            UNIVERSE_MIN_Y) >> 7) + 32;

        Draw_Pixel(sx,sy,12,back_buffer, back_lpitch);
        Draw_Pixel(sx,sy+1,12,back_buffer, back_lpitch);
    }
}

```

```

    } // end if

    } // end for index

// unlock the secondary surface
DDraw_Unlock_Back_Surface();

// draw all the stations
for (index=0; index < MAX_STATIONS; index++)
{
    // draw station blips
    if (stations[index].state==STATION_STATE_ALIVE)
    {
        sx = ((stations[index].varsI[INDEX_WORLD_X] -
            UNIVERSE_MIN_X) >> 7) +
            (SCREEN_WIDTH/2) - ((UNIVERSE_MAX_X -
            UNIVERSE_MIN_X) >> 8);
        sy = ((stations[index].varsI[INDEX_WORLD_Y] -
            UNIVERSE_MIN_Y) >> 7) + 32;

        // test for state
        if (stations[index].anim_state == STATION_SHIELDS_ANIM_ON)
        {
            stationsmall.curr_frame = 0;
            stationsmall.x = sx - 3;
            stationsmall.y = sy - 3;
            Draw_BOB(&stationsmall,lpddsback);
        } // end if
        else
        {
            stationsmall.curr_frame = 1;
            stationsmall.x = sx - 3;
            stationsmall.y = sy - 3;
            Draw_BOB(&stationsmall,lpddsback);
        } // end if
    } // end if
} // end for index

// unlock the secondary surface
DDraw_Lock_Back_Surface();

// draw player as white blip
sx = ((int(player_x) - UNIVERSE_MIN_X) >> 7) + (SCREEN_WIDTH/2) -
    ((UNIVERSE_MAX_X - UNIVERSE_MIN_X) >> 8);
sy = ((int(player_y) - UNIVERSE_MIN_Y) >> 7) + 32;

int col = rand()%256;

```

```

Draw_Pixel(sx,sy,col,back_buffer, back_lpitch);
Draw_Pixel(sx+1,sy,col,back_buffer, back_lpitch);
Draw_Pixel(sx,sy+1,col,back_buffer, back_lpitch);
Draw_Pixel(sx+1,sy+1,col,back_buffer, back_lpitch);

// unlock the secondary surface
DDraw_Unlock_Back_Surface();

// now draw the art around the edges
hud.x      = 320-64;
hud.y      = 32-4;
hud.curr_frame = 0;
Draw_BOB(&hud,lpddsback);

hud.x      = 320+64-16;
hud.y      = 32-4;
hud.curr_frame = 1;
Draw_BOB(&hud,lpddsback);

hud.x      = 320-64;
hud.y      = 32+128-20;
hud.curr_frame = 2;
Draw_BOB(&hud,lpddsback);

hud.x      = 320+64-16;
hud.y      = 32+128-20;
hud.curr_frame = 3;
Draw_BOB(&hud,lpddsback);

} // end Draw_Scanner

```

我想让你看一个典型的扫描器函数，因为这些代码有些混乱。扫描器代表游戏中不同物体的位置，通常是可测的或处于中间。问题在于把一个大空间变成一个小空间，绘制看起来真实的图像元素。这样扫描器的图像通常就由许多不同类的图像元素组成。

当然，当你在浏览一个扫描器时，你想很快的挑选重要数据如你的位置、敌人的方位等，因此颜色和形状非常重要。最后，我决定用一个或多个像素代表敌人，浅灰色代表行星，实际的位图代表边哨。游戏者的飞船用发光斑点代替。

最后，扫描器本身被认为是某种全息图像系统。为了使它看起来效果更好，我在转角处画了一些好看的位图。

至于扫描器的工作算法，看一下代码就可以了。它只不过通过一些常量把每个物体的位置分开了，所以很适合扫描器的窗口。

微粒系统

用于 Outpost 的微粒系统确实很像第十三章“基本物理建模”。粒子可以用各种速度和颜色来创建，可以用一些函数创建模拟爆炸等特殊效果的粒子。最重要的不是这些粒子怎样工作（因为你已经知道），而是怎样使用它们。

在 Outpost 中粒子用于很多地方。我希望所有的敌人在毁掉时都留下水气尾迹。我希望游戏者在飞行时留下等离子体。当某个东西毁掉或放大时，根据位图爆炸的情形我希望有许多粒子，它们作为爆炸的一部分。

关于粒子比较酷的事情就是它们十分简单，却大大的增加了游戏的刺激性。另外水气尾迹和粒子还可以作为游戏元素本身，例如用于追踪或其他的用处：食物、足迹等等。

执行游戏

执行游戏很简单：你只要到处飞行并把物体击中。然而如果说游戏的目的，那就是毁掉所有的边哨。

下面是控制方法：

左右方向键	飞船旋转
向上方向键	推进
Ctrl+空格键	发射武器
H	记录战术信息
S	查看扫描器
左 Alt+右 Alt+A	特别用处
Esc	退出

编译 Outpost

编译 Outpost 同编译其他你创建的演示程序没有什么区别。图 14.17 说明了需要编译和运行的工程。

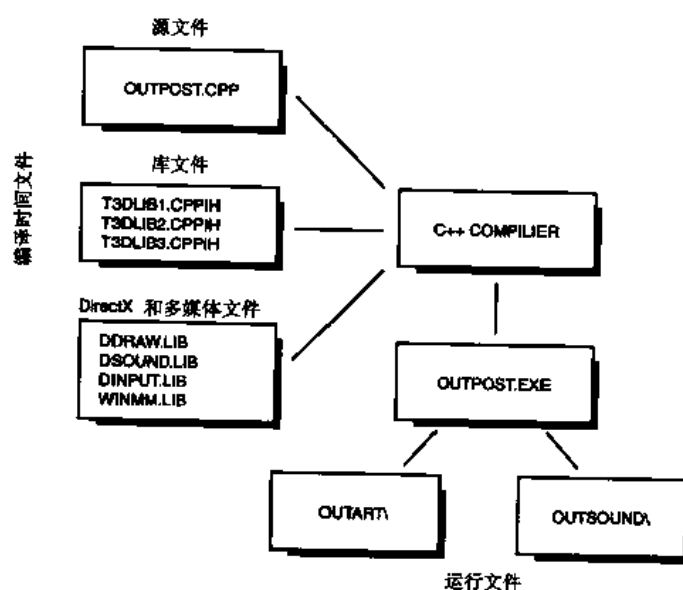


图 14.17 Outpost 的软件结构

详细地看一下接下来的这些组件。

编译文件

源文件:

OUTPOST.CPP	主程序文件
T3DLIB1.CPP/H	游戏工程的第一部分
T3DLIB2.CPP/H	游戏工程的第二部分
T3DLIB3.CPP/H	游戏工程的第三部分

库文件

DDRAW.LIB	MS DirectDraw
DSOUND.LIB	MS DirectSound
DINPUT.LIB	MS DirectInput
WINMM.LIB	Win32 多媒体库

技巧



你必须在你的工程中包含进 DirectX 库文件，光设定搜索路径是不够的。除此之外你必须在 DirectX SDK 安装目录上设定搜索路径以找到 Direct.h 头文件。

运行文件

主文件

OUTPOST.EXE

这是游戏主文件。它可以存在于任何地方，
但媒体目标必须在它下面

运行媒体目录:

OUTART\

游戏的图片目录。全部需要

OUTSOUND\

游戏的声音目录。全部需要

当然,在你的系统中需要 DirectX 的可执行文件。最后,所有的 3D 模型都在 OUTMODELS\ 目录下,所以你的自由度很大。

结束语

我不再说了……我们真的做完了吗?真的结束了吗?不,没有。还有第二卷,主要讲 3D 信息、高等物理和真正的高等数学。

然而,本书的配套 CD 中含有很多有关程序的信息,包括 Direct 3D 和普通 3D。不管你是否拥有第二卷,一定要阅读所有的文章和 CD 上 Sergei Savchenko 及 Matthew Ellis 的计算机程序的文章。你可以在内容列表和附录 A 中找到更多的有关 CD 的内容。此外,你还可以检测一下有关资源、C++ 和数学方面的一些用于特色素材的细微的东西。

第四部分

附 录

附录 A

CD 上的内容

附录 B

安装 *DirectX* 和使用 *C/C++* 编译器

附录 C

三角函数和矢量

附录 D

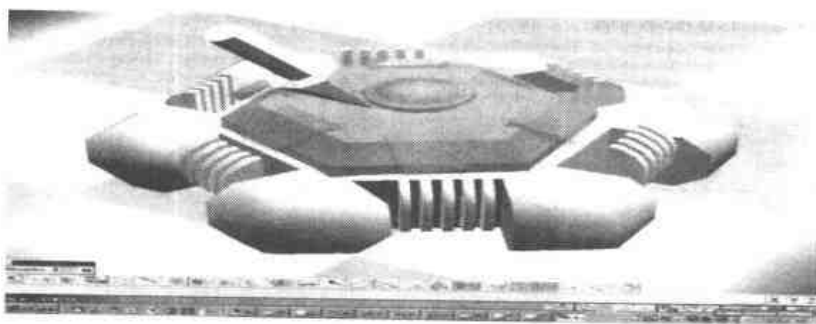
C++ 基础

附录 E

游戏编程资源

附录 F

ASCII 表



A

CD 上的内容

CD-ROM 上含有全部的源程序代码、可执行程序、实例程序、存储技巧、软件程序、音效、在线手册、图形引擎以及补充本书的辅助技术文章。下面是目录结构：

```
CD-DRIVER:\
T3DGAME\
SOURCE\
    T3DCHAP01\
    T3DCHAP02\
    .
    .
    T3DCHAP014\
APPLICATIONS\
ARTWORK\
    BITMAPS\
    MODELS\
SOUND\
    WAVES\
    MIDI\
DIRECTX\
GAMES\
GOODIES\
ARTICLES\
ENGINES\
ONLINEBOOKS\
```

每一个主目录下含有所需要的技术数据。下面是更详细的细目分类：

T3DGAME	含有所有其他目录的根目录。在每次更换目录前，请首先阅读一下 README.TXT 文件。
SOURCE	含有本书所有的按章节排列的源目录。只要将整个 SOURCE\ 目录全部复制到硬盘上，就可以从硬盘上运行。
APPLICATIONS	含有许多公司授权使用的演示程序。
ARTWORK	包含可以在你的程序中免费使用的存储原图。
SOUND	包含可以在你的程序中免费使用的存储的音效和音乐。
DIRECTX	包含最新版本的 DirectX SDK。
GAMES	包含大量的 2D 和 3D 的非常好的共享软件游戏。
GOODIES	包含最后的处理。
ARTICLES	含有游戏编程界的大师们撰写的、利于启发游戏编程灵感的论文。
ENGINES	含有大量的 2D 和 3D 引擎，包括 Genesis 3D 引擎、PowerRender 和 Digital FX 引擎。
ONLINEBOOKS	含有两套完整的数字在线手册，Direct3D 和 General 3D 图形。

提 示

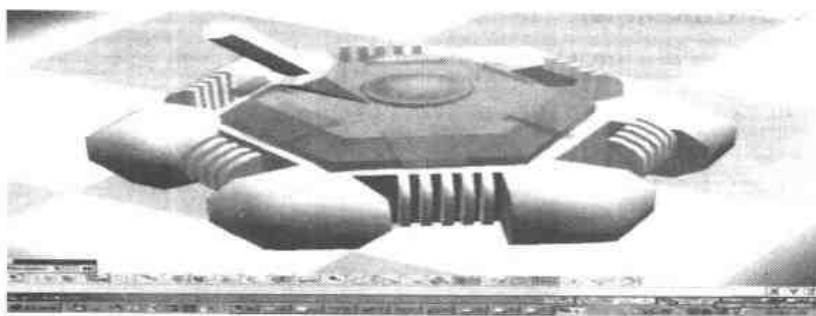
另外，ONLINEBOOK\ 目录下还有完整的 Direct3D 和 General 3D 图形的使用范围说明，因此不要忘记将它们挑拣出来为进入第二卷开个好头。

由于该 CD 包含许多不同类型的程序和数据，因此没有常规安装程序。我把安装程序留给你来解决。但是，在大多数情况下，只要将 SOURCE\ 目录复制到硬盘上，在硬盘上运行就可以了。对于其他程序和数据的情况，当你需要它们时，再安装这些程序和数据。只要将它们拖到硬盘上，在各自的目录中运行各自的安装程序即可。

警 告



当你从 CD-ROM 上复制文件时，要设置多次 ARCHIVE(归档)属性和/或 READ-ONLY(只读)属性。确认复制到硬盘上的文件能够重新设置这些属性。可以通过在 Windows 系统下选择相关的文件和目录，使用快捷键 Ctrl+A 来全部选中，按下鼠标右键，选择文件属性，来重新设置这些属性。一旦出现属性对话框，重新设置 READ-ONLY 和 ARCHIVE 属性，然后按下应用，完成工作。



B

安装 DirectX 和使用 C/C++ 编译器

该 CD 中必须安装的最重要的部分是 DirectX SDK 和运行文件。安装程序在 `DIRECTX\` 目录下，该目录下还有 `README.TXT` 文件，它记录了最后的变动。

注意

使用该 CD 和本书必须应用 DirectX 6.1 SDK 或以上的版本。如果你不能确认系统中是否是最新的文件，运行安装程序，该安装程序可以告诉你。

如果你正在安装 DirectX，注意安装程序放置在 SDK 文件的什么位置。在编译时应当指定编译器的 `LIB` 和 `HEADER` 的合适的搜索路径。

另外，当安装 DirectX SDK 时，安装程序会询问是否想安装 DirectX 运行文件。运行该程序时需要该运行文件以及 SDK。但是运行时间库有两套版本：

测试版

该版本是测试版，我建议安装它用来开发软件。但是 DirectX 程序运行速度要慢一些。

正式版

该版本是完全的用户使用的正式版。该版本要比测试版运行速度快。也可以在安装了测试版之后再安装该版本。

注意

警告 Borland 用户（如果还有的话）：DirectX SDK 也有 DirectX.LIB 输入库函数的 Borland 版本。该版本在 DirectX SDK 安装程序的 `BORLAND\` 目录下。在编译时使用这些文件。并且在使用 Borland 编译器编译 DirectX 程序时，要确认去访问 Borland 的网站，阅读 `BORLAND\` 目录下的 `README.TXT` 文本来了解最后的提示。

最后，到完成该书时，Microsoft 已经推出了 6 个以上的 DirectX 版本。确认经常访问 DirectX 网站以使用 SDK 的最新版本。DirectX 网站是：

<http://www.microsoft.com/directx/>

DirectMedia 是高级的、基于浏览器的 DirectX 系统，可以通过脚本、控件和 Java 来进行控制。实际上也可以在浏览器上创建 DirectX 游戏。在以下地址进行下载：

<http://www.microsoft.com/directx/overview/dxmedia/default.asp>

使用 C/C++ 编译器

在过去的 3 年间我收到了 17000 多封求教如何使用 C/C++ 编译器的电子邮件。我不希望再收到关于编译器问题的电子邮件。每个问题都是编译器用户新手提出的。不阅读使用手册，就不要使用像 C/C++ 编译器一样复杂的软件，好吗？拜托了。因此在根据本书编译程序之前请阅读使用手册。

下面是编译程序的方法：在本书中我使用 MS VC++5.0 和 6.0，因此使用那些编译器可以很好地完成工作。我也希望能够使用 VC++4.0 来进行编译，但是这是不可能的（我听说 DirectX6.0 和 VC++4.0 配合存在许多问题）。如果你使用 Borland 或者是 Watcom 编译器的话，这两种编译器也可以编译，但是要得到编译器的正确的安装程序还要做许多工作。我的建议是不要费尽脑汁了，只要花 99 美元买一套 VC++ 就行了。

MS 编译器对于 Windows/DirectX 程序来讲是最好用的编译器了，它能使一切都更好地运行。我用过 Borland 和 Watcom 的编译器，但对于 Windows 应用程序来讲，我认为不使用 MS VC++ 编译器的专业游戏程序员较少。正确的工作应当使用正确的工具。

下面是安装 MS VC++ 编译器的一些提示。其他的编译器也都类似。

应用程序类型——DirectX 程序都是 Windows 程序。更准确地说，它们都是 Win32.EXE 应用程序。因此，将用于所有 DirectX 程序的编译器设定为 Win32.EXE 应用程序。如果你正在编写控制台应用程序的话，就将编译器设定为控制台应用程序。我建议你创建一个单独的工作空间，在其中编译所有的程序。

搜索目录——要编译 DirectX 程序，编译器需要两个文件：.LIB 文件和.H 文件。请注意：在编译器/连接器选项中设定两个搜索 DirectX SDK 的 .LIB 目录和.H 目录的路径，以便于编译器在编译过程中能够搜索到这些文件。当然这还不够，还必须绝对把握 DirectX 在所有的目录树中位于一级子目录。原因就是 VC++ 是和 DirectX 的老版本同时推出的，如果不仔细的话，将终止连接 DirectX3.0 文件。并且要确认在工程中手动包含了该 DirectX .LIB 文件。我希望在工程中能够看到 DDRAW.LIB、DSOUND.LIB、DINPUT.LIB 等等库文件，这是至关重要的!!!

出错级别设置——确认将编译器中的出错级别以合理的方式返回，如级别 1 和 2。不能关闭该设置，也不能将所有的错误全部提交。本书中我编写的程序代码是专业水平的 C/C++ 程序，编译器认为程序中有大量的我想做但又不必做的事情，因此没有警告级别的错误。

匹配错误——如果编译器在一行代码中出现一个匹配错误（VC++6.0 用户应当注意），

就消除它。我已经收到了 3000 多封不知匹配是什么的读者的电子邮件。如果你也不知道的话, 请在 C/C++ 书中查找一下。但是在我的程序中也仍然不时地出现明显的匹配错误。VC++6.0 最容易出现匹配错误。如果出现了该错误, 看一下编译器希望怎样做, 并将它放在表达式的 rvalue 的前面, 就可以解决。

优化设置——你还不能做出完全成熟的产品来, 因此不要将编译器设置为不安全的优化级别。只要将它设置为速度优先的标准优化设置。

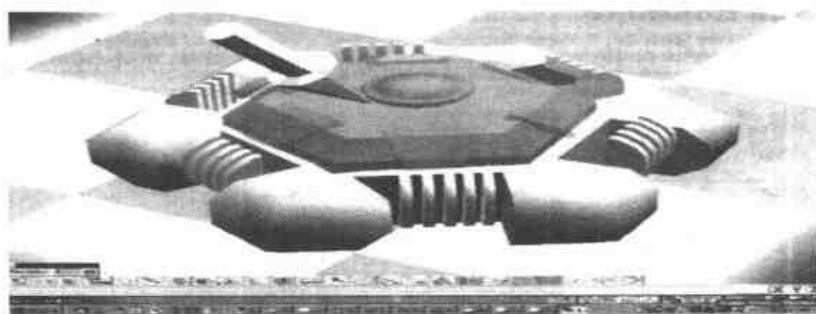
线程模型——本书中 99% 的实例都是单线程的, 因此使用单线程库函数。如果不知道单线程的含义, 请阅读编译器的书籍。我使用多线程库函数时, 我会告诉你的。例如, 要编译第 11 章“算法、数据结构、内存管理及多线程”中的多线程实例时, 必须要转换为多线程库函数。

代码生成——控制编译器生成的代码类型。将它设定为 Pentium 类型。我已经很长时间没有看到 486 了, 因此不必担心兼容性的问题。

结构调整——控制结构的字节。PentiumX 处理器喜欢 32 位的倍数的结构, 因此结构设置应当尽可能高。这样尽管程序长了一点, 但是速度也快了许多。

最后, 当你正在编译程序时, 确认已经包含了程序中访问的所有的源文件。例如, 如果我包含了 T3DLIB1.H 文件, 那就是说在工程中就需要有 T3DLIB1.CPP 源文件。

C



三角函数和矢量

我喜欢数学。你知道为什么吗？因为数学是清楚明白、无可争辩的。做某件事我从来不必考虑最好的解决方法，该怎么样就怎么样。

本附录对数学进行简要的回顾，主要分为两部分内容，可以任意看其中的一部分内容。这仅仅是一个参考，因此读者可以阅读它并说：“好啊，好极了！”

三角

三角是研究角度、斜率及其关系的学问。大部分三角的内容都是建立在对一个直角三角形的分析基础上的，如图 C.1 所示。

表 C.1 列出了弧度/角度值。

表 C.1

弧度/角度值

360 度 = 2π ，即约为 6.28 弧度

180 度 = π ，即约为 3.14159 弧度

$\frac{360 \text{度}}{2\pi} = 1$ 弧度，即约为 57.296 度

$\frac{2\pi}{360}$ 弧度 = 1 度，即约为 0.0175 弧度

下面是三角学中的一些公理：

公理 1：一个完整的圆为 360 度，或者是 2π 弧度。因此， π 弧度为 180 度。计算机中的三角函数 $\sin()$ 和 $\cos()$ 都使用弧度运算，不使用度数，一定要记清楚。

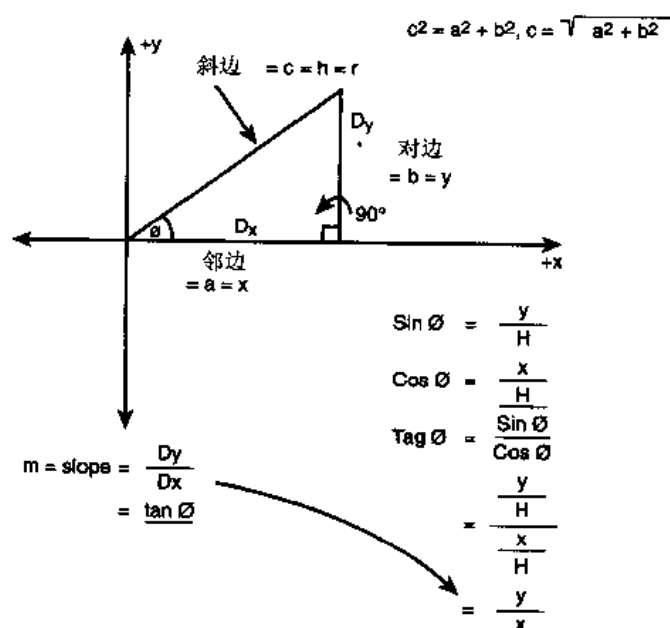


图 C.1 直角三角形

公理 2: 三角形中内角和为 $\theta_1 + \theta_2 + \theta_3 = 180^\circ$ 或者是 π 。

公理 3: 参照图 C.1 的直角三角形, θ_1 角的对边称为对边, θ_1 角的下面的边称为邻边, 三角形中的长边称为斜边。

公理 4: 直角三角形对边和邻边的平方和等于斜边的平方。该公理称为毕达哥拉斯定理 (为我国数学家最先发现。——译注) 用数学符号表示, 该定理可以写为:

$$\text{斜边}^2 = \text{邻边}^2 + \text{对边}^2$$

有时使用 a 、 b 、 c 来作为哑元变量, 可以表示为:

$$c^2 = a^2 + b^2$$

因此说, 如果知道了一个直角三角形的两条边, 就可以求出第三边。

公理 5: 数学家喜欢使用 \sin 、 \cos 和 tag 来表示三个主要的三角比率, 分别定义为:

$$\cos \theta = \frac{\text{邻边}}{\text{斜边}} = \frac{x}{r}$$

定义域: $0 \leq \theta \leq 2\pi$

值域: -1 到 1

$$\sin \theta = \frac{\text{对边}}{\text{斜边}} = \frac{y}{r}$$

定义域: $0 \leq \theta \leq 2\pi$

值域: -1 到 1

$$\text{tag } \theta = \frac{\sin \theta}{\cos \theta} = \frac{\text{对边} / \text{斜边}}{\text{邻边} / \text{斜边}} = \frac{\text{对边}}{\text{邻边}} = \frac{y}{x} = \text{斜率} = M$$

定义域: $-\pi/2 \leq \theta \leq \pi/2$

值域: $-\infty$ 到 ∞

图 C.2 表示了所有函数的图形。几个函数都是周期性的（重复出现）， $\sin \theta$ 和 $\cos \theta$ 函数的周期为 2π ，而 $\tan \theta$ 函数周期为 π 。注意当 θ 趋近于 $\pi/2$ 时， $\tan \theta$ 函数趋近于无穷大。

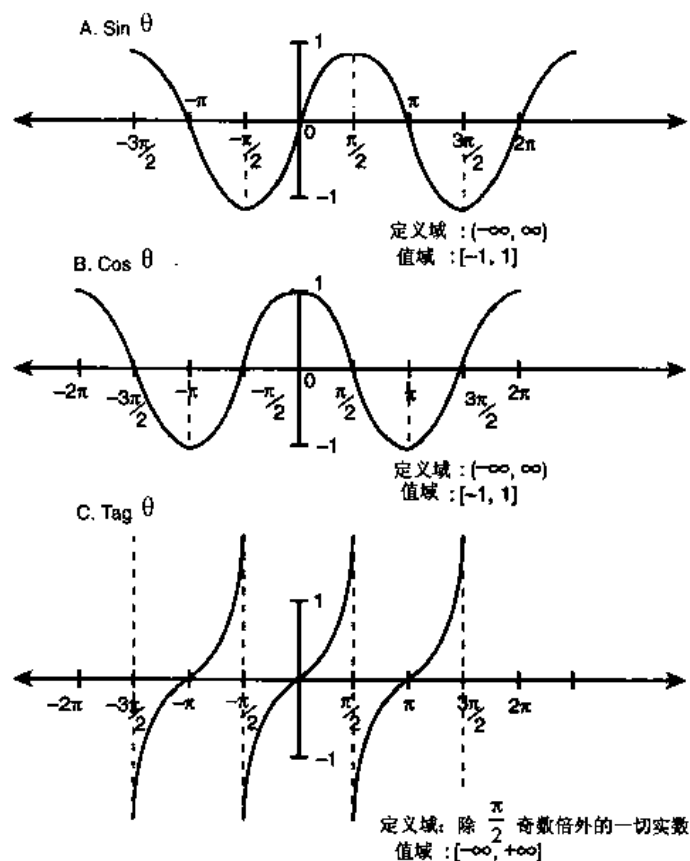


图 C.2 基本三角函数的图形

注意

应当注意定义域和值域名词的使用。定义域和值域分别表示输入和输出。

如果对三角恒等关系不清楚，请阅读数学书。我只准备介绍一些作为游戏程序员应当知道的基本内容。表 C.2 列出了一些三角函数及其关系公式。

表 C.2

常用的三角函数恒等式

Cosecant: $\csc \theta = 1/\sin \theta$ Secant: $\sec \theta = 1/\cos \theta$ Cotangent: $\cot \theta = 1/\tan \theta$

三角函数的毕达哥拉斯定理:

$$\sin^2 \theta + \cos^2 \theta = 1$$

转换恒等式:

$$\sin \theta = \cos(\theta - \pi/2)$$

反射定律:

$$\sin(-\theta) = -\sin \theta$$

$$\cos(-\theta) = \cos \theta$$

加法定律:

$$\sin(\theta_1 + \theta_2) = \sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2)$$

$$\cos(\theta_1 + \theta_2) = \cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2)$$

$$\sin(\theta_1 - \theta_2) = \sin(\theta_1)\cos(\theta_2) - \cos(\theta_1)\sin(\theta_2)$$

$$\cos(\theta_1 - \theta_2) = \cos(\theta_1)\cos(\theta_2) + \sin(\theta_1)\sin(\theta_2)$$

当然, 还可以推导出更多的恒等式来。通常来说, 使用这些恒等式可以简化复杂的三角公式, 而不必去推导数学公式。因此在编程过程中当提出一个基于 \sin 、 \cos 、 \tan 等等三角关系的算法时, 应当翻阅一下三角关系的书籍, 看是否能够简化数学计算, 以便于减少计算结果花费的时间。请记住: 速度、速度、还是速度! 速度是最重要的。

矢量

矢量是游戏程序员最好的朋友。实际上矢量就是由起点和终点定义的线段, 如图 C.3 所示。

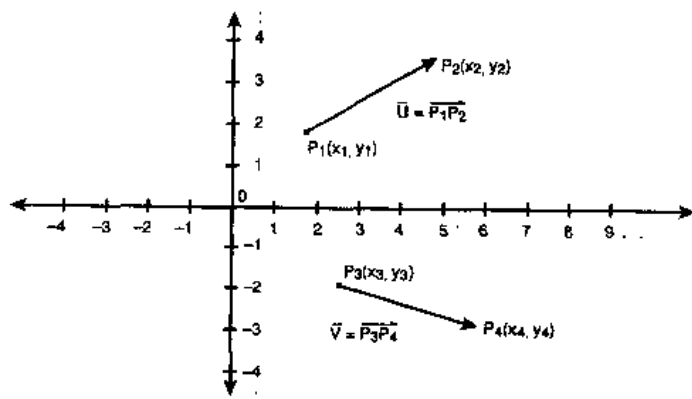


图 C.3 平面矢量

从图 C.3 可以看到由两个点 p_1 (起点) 和 p_2 (终点) 定义的矢量 U 。矢量 $U = \langle u_x, u_y \rangle$ 从点 $p_1(x_1, y_1)$ 指向点 $p_2(x_2, y_2)$ 。要计算矢量 U ，只要从终点位置减去起点位置就可以得到：

$$U = p_2 - p_1 = (x_2 - x_1, y_2 - y_1) = \langle u_x, u_y \rangle$$

通常使用粗体大写字母来表示矢量。分量都写在尖括号中，如： $\langle u_x, u_y \rangle$ 。

矢量表示一条从一点到另一点的线段，而该线段能够表示大量概念，如：速度、加速度以及其他概念。警告：定义的矢量都是相对于原点而言的。也就是说一旦创建了一个从 p_1 到 p_2 的矢量，矢量空间的原点就是 $(0, 0)$ ，在三维空间原点为 $(0, 0, 0)$ 。这无关紧要，因为数学可以解决一切，但是如果认真考虑的话，这是有意义的。

一个矢量在 2D 和 3D 空间中就是两个或三个数字，因此只要在 2D 或 3D 空间中定义一个终点就可以了。也就是说起点一般来讲都采用原点。这并不表示不能任意转化矢量、对矢量进行各种几何运算。这仅仅表示应当记住矢量到底是什么。

关于矢量最酷的是可以对矢量进行各种运算。因为矢量实际上就是按次序排列的数集，可通过分别对构成矢量的各个分量分别进行运算来对矢量进行各种标准数学运算。

注 意

矢量可以有許多分量。通常在计算机图形学中处理 2D 和 3D 矢量，矢量形式一般为 $A = \langle x, y \rangle$ 或者 $B = \langle x, y, z \rangle$ 。一个 n 维的矢量形式是 $C = \langle c_1, c_2, c_3, \dots, c_n \rangle$ 。 n 维矢量用来表示变量集而不是几何空间，因为多于 3D 空间就称为超空间（多维空间）。

矢量长度

使用矢量时碰上的第一件事情就是如何计算矢量的长度。矢量长度称为模，用两个竖条（绝对值符号）来表示，如： $|U|$ ，称为矢量 U 的长度。

矢量长度就是计算从该矢量的起点到终点的距离。因此，可以使用标准毕达哥拉斯定理来求解矢量长度。因此 $|U| = \sqrt{u_x^2 + u_y^2}$

如果 U 是 3D 矢量，其长度为：

$$|U| = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

标准化（归一化）

一旦求得矢量的长度，可以对该矢量进行标准化（归一化），使矢量长度为 1。单位矢量具有许多特性，就像一个标量 1.0 一样，凭直觉我认为你也会同意这种说法。给定一个矢量 $N = \langle n_x, n_y \rangle$ ，矢量 N 的单位矢量使用小写字母 n 表示：

$$n = N / |N|$$

非常简单，单位矢量就是被矢量长度除过的矢量。

标量乘法

对矢量进行的第一种运算就是数乘。矢量的每个分量乘以一个标量数字，如下所示：

设 $\mathbf{U} = \langle u_x, u_y \rangle$

$K \times \mathbf{U} = K \times \langle u_x, u_y \rangle = \langle k \times u_x, k \times u_y \rangle$

图 C.4 表示了数乘运算的示意图。

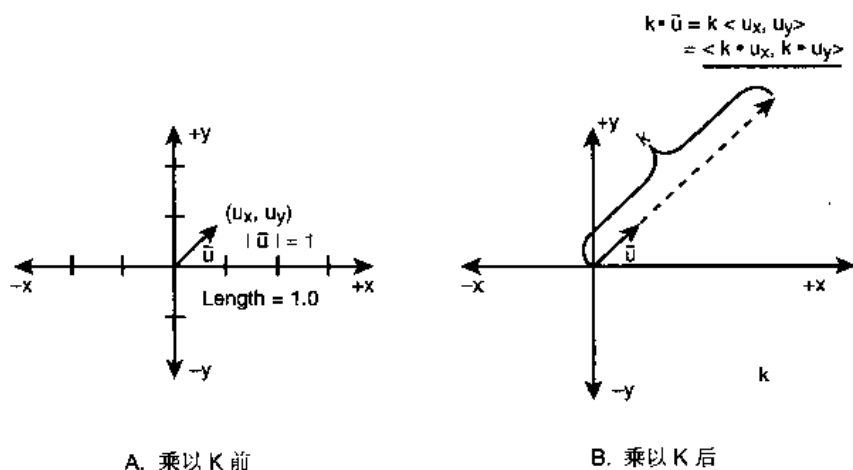


图 C.4 矢量数乘

另外，如果令矢量方向相反，可是使该矢量乘以-1，就可以得到方向相反的矢量，如图 C.5 所示。

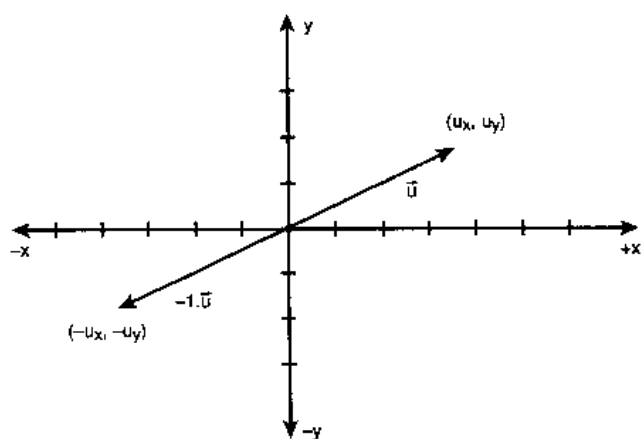


图 C.5 矢量反向

数学表示：

设 $\mathbf{U} = \langle u_x, u_y \rangle$

和矢量 \mathbf{U} 方向相反的矢量为：

$$-1 \times U = -1 \times \langle u_x, u_y \rangle = \langle -u_x, -u_y \rangle$$

矢量加法

要将两个或多个矢量相加，只要将矢量的各分量分别相加就可以了。图 C.6 给出了示意图。

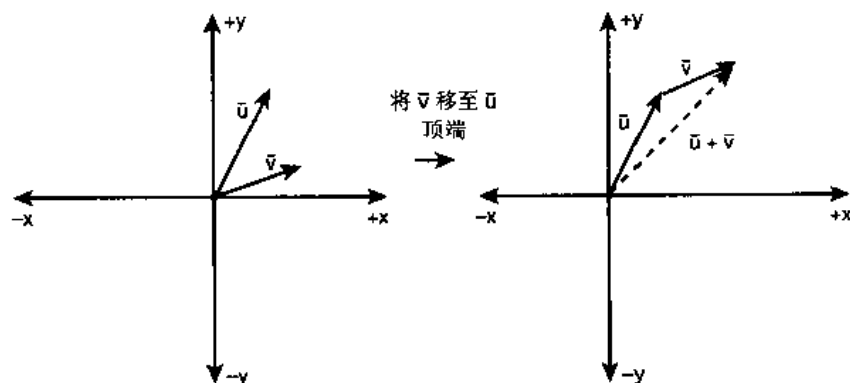


图 C.6 矢量相加

矢量 **U** 加上矢量 **V** 等于矢量 **R**。注意矢量加法运算的过程。将矢量 **V** 移动到矢量 **U** 的终点，然后以相同的方向画一矢量，就可以得到矢量 **R**。使用数学运算表示：

$$U + V = \langle u_x, u_y \rangle + \langle v_x, v_y \rangle = \langle u_x + v_x, u_y + v_y \rangle$$

因此，在方格纸上要将任意个矢量相加，只要将所有的矢量首尾相连，当将所有矢量相加后，从原点到最后一个终点的矢量就是要求的矢量。

矢量减法

矢量相减实际上就是加上一个相反方向的矢量。用图示方法表示矢量相减也是很有意义的。图 C.7 给出了 **U - V** 和 **V - U** 示意图。

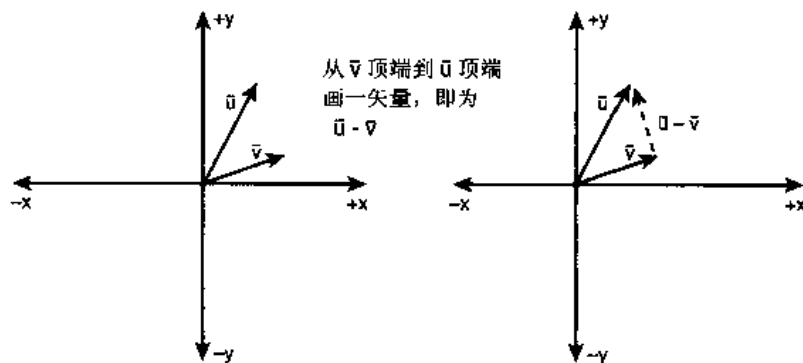


图 C.7 矢量相减

注意： $\mathbf{U}-\mathbf{V}$ 运算就是从矢量 \mathbf{V} 向矢量 \mathbf{U} 画一矢量，而 $\mathbf{V}-\mathbf{U}$ 运算就是从矢量 \mathbf{U} 向矢量 \mathbf{V} 画一矢量，数学运算表示：

$$\mathbf{U}-\mathbf{V} = \langle u_x, u_y \rangle - \langle v_x, v_y \rangle = \langle u_x - v_x, u_y - v_y \rangle$$

非常容易，有时使用人工计算时，在方格纸上进行矢量相减更加方便，因为结果非常直观。所以说在编写算法时，了解如何在方格纸上进行矢量相加减是个好主意——请相信我。

点积（内积）

此时可能你会问：“两个矢量能够相乘吗？”答案是可以相乘，但是事实上直接按分量相乘是没有用的。换句话说：

$$\mathbf{U} \times \mathbf{V} = \langle u_x \times v_x, u_y \times v_y \rangle$$

该式并不能表示矢量空间中的任何东西。但是可以定义矢量的点积，如下所示：

$$\mathbf{U} \cdot \mathbf{V} = u_x \times v_x + u_y \times v_y$$

点积通常使用一个点(\cdot)来表示，通过两个矢量的各自分量乘积的和来求得。当然结果是获得一个标量。但是这又有什么用？得到的结果已经不是矢量了。矢量的点积等于下面表达式：

$$\mathbf{U} \cdot \mathbf{V} = |\mathbf{U}| \times |\mathbf{V}| \times \cos \theta$$

该式表示 \mathbf{U} 点积 \mathbf{V} 等于 \mathbf{U} 的模乘以 \mathbf{V} 的模再乘以该两个矢量的夹角的余弦。如果将两个公式合并的话，可以得到：

$$\mathbf{U} \cdot \mathbf{V} = u_x \times v_x + u_y \times v_y$$

$$\mathbf{U} \cdot \mathbf{V} = |\mathbf{U}| \times |\mathbf{V}| \times \cos \theta$$

$$u_x \times v_x + u_y \times v_y = |\mathbf{U}| \times |\mathbf{V}| \times \cos \theta$$

这是一个非常有意思的公式，首先给出了一种计算两个矢量夹角的公式，如图 C.8 所示，矢量的点积的确是一个非常有用的运算。

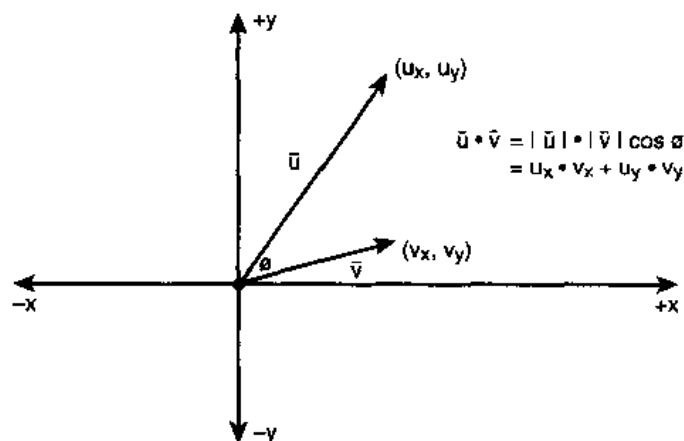


图 C.8 点积

如果对此还不理解的话，对上面公式重新排列，然后对两边取 \cos 的反函数：

$$\theta = \cos^{-1}((u_x \times v_x + u_y \times v_y) / (|U| \times |V|))$$

或者更严密一点的话，使用 $U \cdot V$ 来代替 $(u_x \times v_x + u_y \times v_y)$ ，有：

$$\theta = \cos^{-1}(U \cdot V / (|U| \times |V|))$$

这是一个非常强大的工具，也是许多 3D 图形算法的基础。最酷的是如果 U 和 V 的模均为 1 的话，其点积也为 1，公式可以进一步简化为：

$$\theta = \cos^{-1}(U \cdot V), |U|=|V|=1.0$$

下面是一些有用的公理：

公理 1：如果 U 和 V 的夹角为 90 度的话， $U \cdot V=0$ 。

公理 2：如果 U 和 V 的夹角小于 90 度（锐角）的话， $U \cdot V>0$ 。

公理 3：如果 U 和 V 的夹角大于 90 度的话， $U \cdot V<0$ 。

公理 4：如果 U 和 V 相等的话， $U \cdot V=|U|=|V|=1.0$ 。

图 C.9 为以上公理的图示。

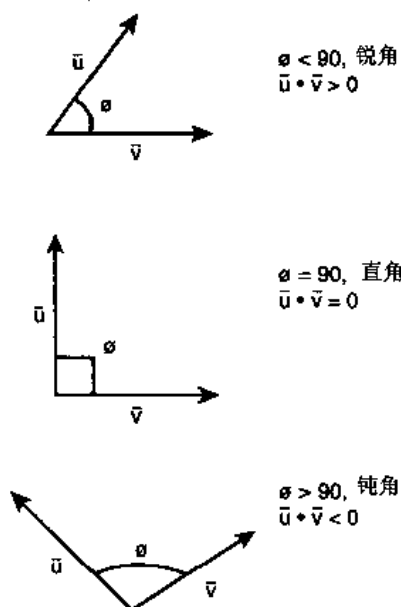


图 C.9 夹角及其与点积的关系

叉积（矢量积）

另外一种矢量的乘积是矢量积。矢量积仅仅对于有三个以上分量的矢量有意义，因此我们使用 3D 空间矢量来作为实例。给定矢量 $U=\langle u_x, u_y, u_z \rangle$ ， $V=\langle v_x, v_y, v_z \rangle$ ，矢量积写为 $U \times V$ ，定义如下：

$$U \times V = |U| \times |V| \times \sin \theta \times \mathbf{n}$$

好，下面我们分步分解。 $|U|$ 表示 U 矢量的长度， $|V|$ 表示 V 矢量的长度， $\sin \theta$ 是两个

矢量的夹角的正弦。 $|\mathbf{U}| \times |\mathbf{V}| \times \sin \theta$ 是一个标量，也就是一个数。所以可以对该标量乘以 \mathbf{n} 。但是 \mathbf{n} 是什么？ \mathbf{n} 是一个单位矢量，这也是为什么用小写字母表示的原因。另外， \mathbf{n} 是一个法向矢量，也就是说，它垂直于 \mathbf{U} 和 \mathbf{V} 。图 C.10 给出了其示意图。

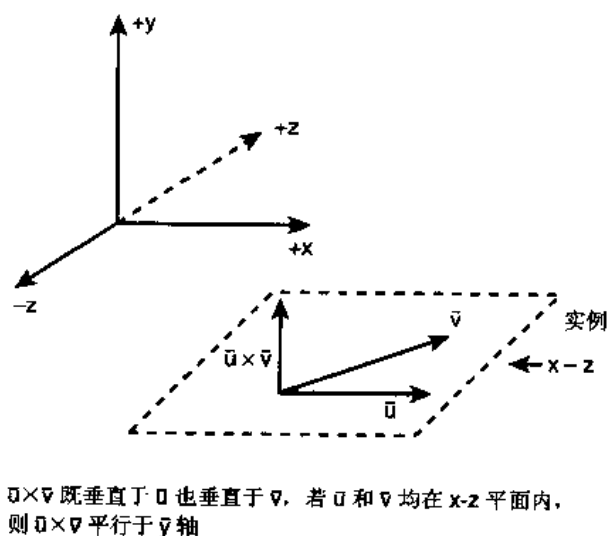


图 C.10 矢量积

矢量积给出了 \mathbf{U} 和 \mathbf{V} 的角度和法向矢量。但是如果没有其他公式，就得不到任何东西。问题是如何计算 \mathbf{U} 和 \mathbf{V} 的法向矢量，以便于能够计算 $\sin \theta$ 项或其他项。矢量积也可以作为一个非常特殊的矢量乘积来定义。但是必须借助于矩阵来表示，请稍候。假定要计算 \mathbf{U} 和 \mathbf{V} 的矢量积 $\mathbf{U} \times \mathbf{V}$ ，首先应当建立一个矩阵，如下所示：

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

其中， \mathbf{i} 、 \mathbf{j} 、 \mathbf{k} 分别是平行于 x 、 y 、 z 轴的单位矢量。

然后，要计算 \mathbf{U} 和 \mathbf{V} 的矢量积，还要进行下面运算：

$$\mathbf{N} = (u_y \times v_z - u_z \times v_y) \times \mathbf{i} + (-u_x \times v_z + u_z \times v_x) \times \mathbf{j} + (u_x \times v_y - u_y \times v_x) \times \mathbf{k}$$

\mathbf{N} 是三个标量与各自的平行于 x 、 y 、 z 轴的正交单位矢量的线性组合，因此，去掉 \mathbf{i} 、 \mathbf{j} 、 \mathbf{k} ，可以将公式写为：

$$\mathbf{N} = (u_y \times v_z - u_z \times v_y, -u_x \times v_z + u_z \times v_x, u_x \times v_y - u_y \times v_x)$$

\mathbf{N} 是 \mathbf{U} 和 \mathbf{V} 的法向矢量，但不一定是一个单位矢量（ \mathbf{U} 和 \mathbf{V} 是单位矢量的话， \mathbf{N} 也是单位矢量），因此必须对 \mathbf{N} 进行标准化，求得 \mathbf{n} 。这样就可以代入到矢量积公式中进行运算。

实际上, 尽管如此, 还是没有几个人使用过 $\mathbf{U} \times \mathbf{V} = |\mathbf{U}| \times |\mathbf{V}| \times \sin \theta \times \mathbf{n}$ 公式。一般都使用矩阵形式来求解法向矢量。法向矢量对于 3D 图形非常重要, 在第二卷中我们将计算大量的法向矢量。法向矢量的重要性不仅仅在于和两个矢量都垂直而具有重要作用, 而且还在于它经常用来定义平面, 计算多边形的方向——这对于碰撞、检测、着色、采光等等又很有用。

零矢量

尽管可能没有使用过零矢量, 但它确实存在。零矢量表示长度为零并且无方向。如果想了解得更详细一点的话, 零矢量就是一个点。因此在 2D 空间中零矢量为 $\langle 0, 0 \rangle$, 在 3D 空间中零矢量为 $\langle 0, 0, 0 \rangle$, 对于更高的维数, 以此类推。

位置矢量

下面我们讨论一下位置矢量。位置矢量常用于描绘几何实体, 如线、线段、曲线等等。我经常在裁剪时使用位置矢量, 在第十三章“基本物理建模”中的线段相交的计算中使用位置矢量, 因此位置矢量是非常重要的。图 C.11 显示了一个用来表示线段的位置矢量。

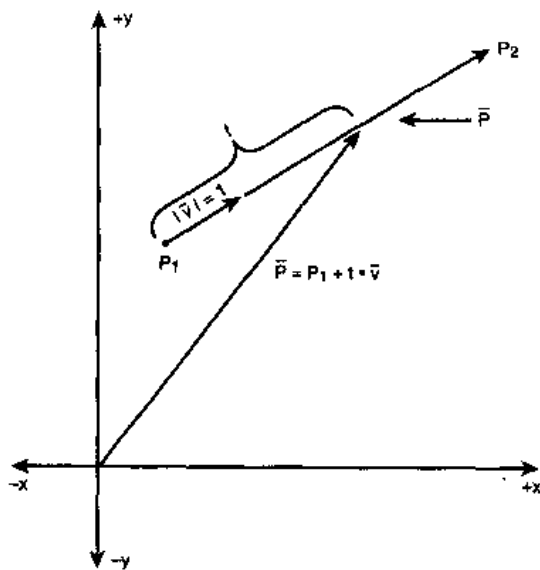


图 C.11 位置矢量

该线段从 p_1 到 p_2 , \mathbf{V} 是从 p_1 到 p_2 的矢量, \mathbf{v} 是从 p_1 到 p_2 的单位矢量。然后构造一个描绘该线段的 \mathbf{P} , 如下所示:

$$\mathbf{P} = \mathbf{p}_1 + t \times \mathbf{v}$$

其中, t 是从 0 到 $|\mathbf{V}|$ 的变量。如果 $t=0$, 有:

$$\mathbf{P} = \mathbf{p}_1 + t \times \mathbf{v} = \langle p_1 \rangle = \langle p_{1x}, p_{1y} \rangle$$

因此，当 $t=0$ 时， \mathbf{P} 是指向线段起点的矢量，而当 $t=|\mathbf{V}|$ 时，有：

$$\mathbf{P}=\mathbf{p1}+|\mathbf{V}|\times\mathbf{v}=\mathbf{p1}+\mathbf{V}=\langle\mathbf{p1}+\mathbf{V}\rangle=\langle\mathbf{p1}_x+\mathbf{v}_x, \mathbf{p1}_y+\mathbf{v}_y\rangle=\mathbf{p2}=\langle\mathbf{p2}_x, \mathbf{p2}_y\rangle$$

此时 \mathbf{P} 是指向线段终点的矢量。

矢量的线性组合

从矢量积运算中可以看到，矢量可以用下面方法表示：

$$\mathbf{U}=\mathbf{u}_x\times\mathbf{i}+\mathbf{u}_y\times\mathbf{j}+\mathbf{u}_z\times\mathbf{k}$$

其中， \mathbf{i} 、 \mathbf{j} 、 \mathbf{k} 分别是平行于 x 、 y 、 z 轴的单位矢量。这并不难于理解，仅仅是矢量的另外一种写法而已。所有的运算都严格地以相同方式计算。例如：

$$\text{令 } \mathbf{U}=3\mathbf{i}+2\mathbf{j}+3\mathbf{k}$$

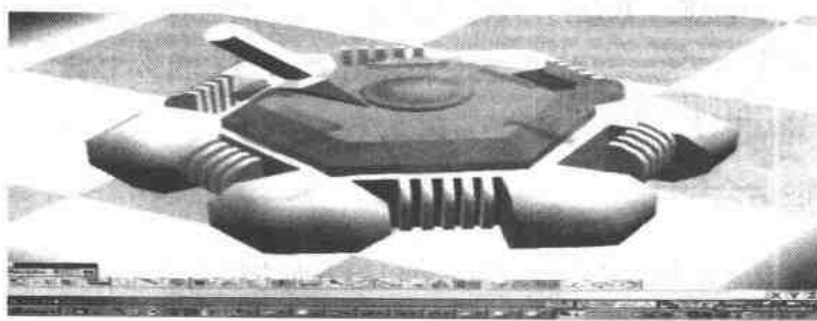
$$\text{令 } \mathbf{V}=-3\mathbf{i}-5\mathbf{j}+12\mathbf{k}$$

$$\mathbf{U}+\mathbf{V}=3\mathbf{i}+2\mathbf{j}+3\mathbf{k}-3\mathbf{i}-5\mathbf{j}+12\mathbf{k}=0\mathbf{i}-3\mathbf{j}+15\mathbf{k}=\langle 0, -3, 15 \rangle$$

这只是一种表示法而已。把矢量作为各分量的线性组合的写法最大的好处是只要每个分量具有矢量系数，该分量就决不会混淆。因此可以编写非常长的表达式，并且对该矢量进行合并同类项。

上面就是关于数学的回顾，请再阅读一遍。

D



C++ 基础

提示

如果你是一个 C++ 程序员，你可能会问：“为什么 Andre 一直使用 C 语言？”答案是使用 C 语言简单，因为 C 语言容易理解，使用方便。C++ 程序员都了解 C 语言，因为 C 语言是 C++ 的子集，并且大多数游戏程序员在学习 C++ 之前都学习过 C 语言。

C++ 是什么

C++ 就是使用面向对象（OO）技术升级的 C 语言。实际上，C++ 不仅仅是 C 语言的扩展集。C++ 主要具有下面升级的内容：

- 类
- 继承性
- 多态性

下面迅速浏览一下每项内容。类仅是合成数据和函数的一种方式。正常情况下，当使用 C 语言编程时，必须具有存放数据的数据结构以及操作数据的函数，如图 D.1 中的 A 部分所示。但是使用 C++，数据和操作数据的函数都存放于一个类中，如图 D.1 中的 B 部分所示。这样做有什么好处呢？可以将类看作是一个具有属性以及能够运行的对象。这仅仅为了思考更方便而已。

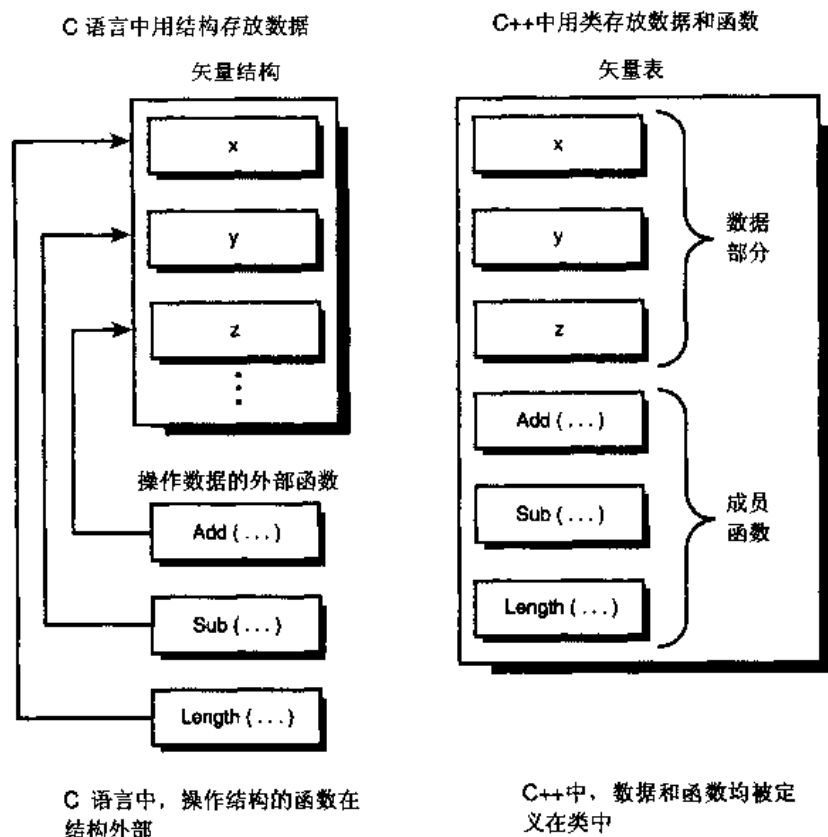


图 D.1 类的结构

C++的另一个特征是继承性。一旦创建了类，就可以建立类对象和基本对象或基于另一个的类之间的关系。实际就是一直这样操作的，那为什么在软件中不这样呢？例如，有一个人（person）的类，含有关于人以及一些操作数据（现在不考虑该内容）的类方法的数据。

人是相当通用的数据。但是继承性在想创建两个不同类型的人（如软件工程师和硬件工程师）的时候就起作用了，我们分别给这两类人命名为 `sengineer` 和 `hengineer`。

图 D.2 给出了人、软件工程师和硬件工程师之间的关系。看该两种新类如何在人的基础上建立？软件工程师和硬件工程师都是人，但是都各有自己的外延数据。所以说创建软件工程师和硬件工程师就继承了人的特性，并且增加了一些新的性质。这是继承的基础。可以从已有的对象建立更复杂的对象。另外，还有多重继承，可以建立一个新对象作为一个子类。

C++和面向对象编程技术的第三个特征是多态性，即“多种状态”。在C++内容中，多态性表示函数或操作符根据其状态不同而不同。例如，在C语言中 $(a+b)$ 表达式表示将 `a` 和 `b` 加在一起，并且 `a` 和 `b` 必须具有类型，如 `int`、`float`、`char` 和 `short`。在C语言中，不能定义一个新类型叫 $(a+b)$ 。但是在C++中就可以这样做。因此，可以重复定义操作符，如 `+`、`-`、`*`、`/`、`[]` 等等，并且根据不同的数据进行不同的运算。

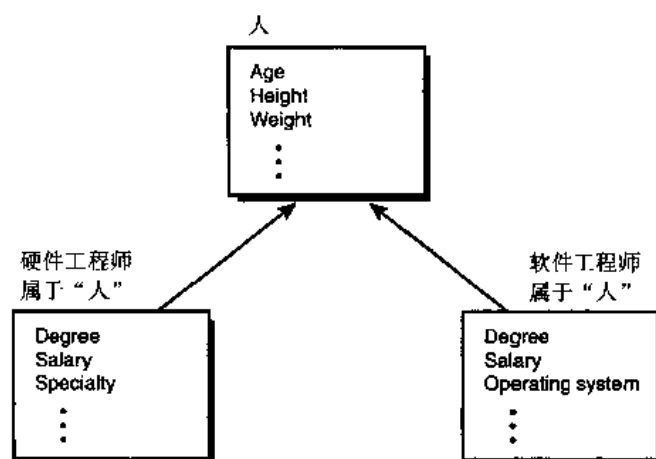


图 D.2 类的继承性

甚至还可以重复定义函数。例如，编写一个函数 `Compute()`，如下：

```
int Compute(float x, float y)
{
    // code
}
```

该函数采用了两个浮点型，但是如果将整数传递给该函数，必须将该整数转换为浮点数，然后再传递给该函数。这样可能会丢失数据。在 C++ 中就可以这样做：

```
int Compute(float x, float y)
{
    // code
}
int Compute(int x, int y)
{
    // code
}
```

尽管两个函数具有相同的名称，但是它们采用了不同的类型。编译器将这两个函数看作是完全不同的两个函数，因此调用整数时就表示调用第二个函数，而调用浮点数时就表示调用第一个函数。但如果使用一个整数和一个浮点数来调用该函数就有点复杂。提升规则开始发挥作用，编译器根据这些新规则来决定调用哪一个函数。

上述就是 C++ 的升级内容。C++ 中还有其他的一些添加的语法和大量的规则，但是对于 C++ 中的大部分内容都是需要实现上面的三个概念。对非常复杂的概念进行了极其简单的解释，是吧！

关于 C++ 应当了解的内容

C++ 是一种极其复杂的语言，使用太多太快的新技术可能编写出完全不可靠的程序来，可能造成内存不足、性能问题等等。使用 C++ 的问题是，这是一种黑盒子式的语言。大量的运行过程都在背后进行，可能发现不了已经存在的错误。但是如果开始的时候经常只使用一点 C++ 语言的内容，当需要时就在程序中添加一点新特征，这样就会掌握 C++。

我编写关于 C++ 内容的附录惟一的原因是 DirectX 是建立在 C++ 语言基础上的。但是 C++ 的大部分内容密封于封装程序和通过函数指针调用进行联系的 COM 界面中，也就是说，格式 `interface->function()` 的调用。如果你已经完全掌握了该内容，那么只需要解决古怪的语法就可以了。COM（组件对象模型）内容的一章有助于解决古怪的语言的问题。不管怎么说，我应当介绍一些基础理论，以便于能够更好地理解 C++、和朋友们讨论 C++ 以及掌握有用的、便于更好工作的知识。

下面我将介绍一些新的类型、协议、内存管理、I/O 流、基本类、函数和操作符重载等内容。

新类型、关键词以及协议

首先以简单的内容——新的注释操作符（`//`）开始。这是 C 语言的一部分，可能已经使用过该操作符，但是 `//` 操作符在 C++ 中是一个单行注释。

注释

```
// this is a comment
```

也可以使用原来的注释样式 `/* */`：

```
/* a C style multi line comment
every thing in here is a comment
*/
```

常量

要使用标准 C 语言创建一个常量，必须：

```
#define PI 3.14
```

或者：

```
float PI=3.14;
```

使用第一种方法存在的问题是 PI 不是一个具有类型的实变量，而只是预处理程序用来进行文本代换的一个符号，因此既没有类型，也没有大小。第二种类型定义方法的问题是其所写因而易改变。因此，C++具有了一个新类型，称之为 const，如同一个只读变量：

```
const float PI=3.14;
```

现在就可以在任何想使用 PI 的地方使用了，PI 的类型是 float，大小为 sizeof(float)，但是不能修改。这是一种定义常量的更好的方法。

参变量

在 C 语言中，常常要改变函数中某个变量的值，可以通过一个指针来完成：

```
int counter = 0;
void foo(int *x)
{
    (*x)++;
}
```

如果调用 foo(&counter)，则 counter 在调用一次后等于 1。函数将改变被传递变量的值。这只是一般的技术，C++ 已经具有了一种新类型的变量，协助完成该项工作，称为参变量，使用地址标识符 & 来表示。

```
int counter = 0;
void foo(int &x)
{
    x++;
}
```

很有意思吧？怎样调用该函数呢？应当：

```
foo(counter);
```

注意，不再需要将 & 放在计数器的前面。x 变成了传递变量的假名。因此计数器是 x，在调用中不再使用 &。

也可以像下面一样创建函数外的参变量：

```
int x;
int &x_alias= x;
```

x_alias 就是 x 的假名。无论怎样使用 x，都可以使用 x_alias，二者是相同的。但我认为并不需要该语法。

任意创建变量

关于 C++ 最酷的一个特征就是能够在代码块中创建变量，而不止是作为函数的全局变量。例如，下面是如何使用 C 语言编写一个循环体：

```
void Scan(void)
{
    int index;
    // lots of code here...
    // finally our loop
    for (index = 0; index<10, index++)
        Load_Data(index);
    //more code here...
} // end Scan
```

该程序没有任何错误，但是 `index` 在一个程序代码段中仅能用于一个循环 `index`。C++ 设计人员认为这样并不很完整，并且认为变量应当在最接近于使用该变量的位置来定义。用于一个代码块中的变量就不能用于其他代码块。例如，如果编写了类似下面的程序块：

```
void Scope(void)
{
    int x = 1, y = 2; // global scope
    printf("\nBefore Block A: Global Scope x=%d, y=%d, x, y);
    { // Block A
        int x=3, y=4;
        printf("\nIn Block A: x=%d, y=%d, x, y);
    } // end Block A
    printf("\nAfter Block A: Global Scope x=%d, y=%d, x, y);
    { // Block B
        int x=5, y=6;
        printf("\nIn Block B: x=%d, y=%d, x, y);
    } // end Block B
    printf("\nAfter Block B: Global Scope x=%d, y=%d, x, y);
} // end Scope
```

上面程序中有三种不同的 `x`、`y` 的版本。定义的第一个 `x`、`y` 是全局变量。一旦进入代码块 A，`x` 和 `y` 就变成代码块中的局部变量。当代码块 A 结束时，原来的 `x` 和 `y` 又恢复成它们原来的值，同理，代码块 B 中的变量变化和 A 相同。在代码块范围内，可以更好地定位和使用变量，而不必一直考虑必须使用新的变量名；可以继续使用 `x` 和 `y` 或其他变量，不必担心新变量是否因和全局变量同名而发生冲突。

关于这种新变量定义范围的好处是可以在代码中任意创建变量。例如，请看下面使用 C++ 编写的关于 `index` 的相同的 `for()` 循环：

```
// finally our loop
for (int index = 0; index<10, index++)
```

```
Load_Data(index);
```

这样做难道不是最酷的吗？我在使用 `index` 时才定义它，而不是在函数的最前面定义 `index`。应当对变量进行紧密的控制。

内存管理

C++ 具有一个基于操作符 `new` 和 `delete` 的新的内存管理系统。在很大程度上它们和 `malloc()` 和 `free()` 相当，但是更敏锐，因为它们考虑了被请求和被删除的数据的类型。实例如下：

使用 C 语言从堆栈中分配 1000 个整数：

```
int *x = (int*)malloc(1000*sizeof(int));
```

多么混乱啊！下面是使用 C++ 编写的代码：

```
int *x = new int(1000);
```

这样做好多了吧！`new` 已经知道了传递回一个指向整数的指针，即 `int*`，因此就不必再指派了。要使用 C 语言释放内存，应当：

```
free(x);
```

而用 C 语言释放内存，应当：

```
delete(x);
```

这一点它们基本相同，但是 `new` 操作符要好一些。使用 C 或 C++ 都可以分配内存。但是不要混合，如使用 `new` 调用，然后再使用 `free()` 调用，或者使用 `malloc()` 调用，然后再使用 `delete` 调用都不可取。

输入/输出流

我喜欢使用 `printf()`，非常清楚：

```
printf("\nGive me some sugar baby.");
```

使用 `printf()` 存在的惟一的问题是需要制定所有的格式标识符，如 `%d`、`%x`、`%u` 等等，非常难记。另外 `scanf()` 就更麻烦了，因为如果忘记了使用存储变量的地址，就会将它们弄乱。例如：

```
int x;
scanf("%d", x);
```

这样就不正确了。应当使用 `x` 或 `&x` 的地址，因此正确的语法是：

```
scanf("%d", &x);
```

我敢说你肯定犯过这样的错误！只有在使用字符串时才不需要使用地址操作符，因为字符串名就是地址。无论怎样说这都是 C++ 中创建 `IOSTREAM` 新类的原因。该类知道变量的类型，因此根本就不需要告诉该类。`IOSTREAM` 类函数在 `IOSTREAM.H` 中定义，因此要使用它必须将它包含于 C++ 程序中。然后就可以访问流 `cin`、`cout`、`cerr` 和 `cprn`，如表 D.1 所示。

表 D.1 C++ 的 I/O 流

流的名称	设备	C 名称	含义
<code>Cin</code>	键盘	<code>stdin</code>	标准输入
<code>cout</code>	屏幕	<code>stdout</code>	标准输出
<code>Ceer</code>	屏幕	<code>stderr</code>	标准出错
<code>cprn</code>	打印机	<code>stdprn</code>	打印机

使用 I/O 流有点古怪，因为它们是基于重载操作符 `<<` 和 `>>` 建立的。这些符号在 C 语言中通常表示位移位，但是在 I/O 流的内容中，它们用来传递和接收数据。下面是使用标准输出的几个实例：

```
int i;
float f;
char c;
char string[80];
// in C
printf("\nHello world!");
// in C++
cout<<"\nHello world!";
// in C
printf("%d", i);
// in C++
cout<< i;
// in C
printf("%d, %f, %c, %s", i, f, c, string);
// in C++
cout<<i<<" "<< f<<" "<<c<<" "<< string;
```

这样做非常酷吧！根本不需要类型标识符，因为 `cout` 知道类型以及要做的工作。该语

法惟一古怪的是为什么 C++ 允许在每一次操作后再串联一个<<操作符。原因是每次操作返回字符串本身，因此要添加上一个<<操作符。使用流进行简单打印的惟一的坏处是必须将变量和字符串常数分开，如使用“,” 隔开每个变量。但是也可以将<<放在每一行上，如：

```
cout <<i
    <<" ,"
    << f
    <<" ,"
    <<c
    <<" ,"
    << string;
```

请记住，在 C 语言和 C++ 中，禁止使用空格，因此这样编程是合法的。输入流和输出流工作原理相似，但是使用>>操作符。下面是几个实例：

```
int i;
float f;
char c;
char string[80];
// in C
printf("\nWhat is your age?");
scanf("%d", &i);
// in C++
cout<<"\n What is your age?";
cin >> i;
// in C
printf("\nWhat is your name and grade?");
scanf("%s %c", &string, &c);
// in C++
cout<<"\nWhat is your name and grade?";
cin>>string>>c;
```

比 C 语言要稍好一些，是吗？当然，Iostream 系统还有一百万个其他的函数，请检验一下。

类

类是 C++ 中最重要的补充，使用语言是面向对象的技术。如我前面所述，一个类仅仅是一个数据和操作数据的方法（通常称为构件函数）的集合体。

C++ 中的新的数据结构

下面我们学习类，首先看一下稍稍有点变化的标准结构，使用 C 语言，可以定义

下面结构：

```
struct Point
{
    int x, y;
};
```

然后，创建一个结构的实例：

```
struct Point p1;
```

这样就创建了结构 `Point` 的一个实例或对象，名称是 `p1`。使用 C++，创建实例就不必使用 `struct` 关键词：

```
Point p1;
```

同样也创建了结构 `Point` 的一个实例，名称是 `p1`。这是因为 C 程序员已经创建了类型，因此就不必再输入 `struct`，如：

```
typedef struct Point_tag
{
    int x, y;
} Point;
```

语法是：

```
Point p1;
```

类和新结构相似，因此就不必创建一个类型。定义本身就是类型。

一个简单的类

C++中的类使用关键词 `class` 来定义。例如：

```
class Point
{
public:
    int x, y;
};
Point p1;
```

这几乎和 `Point` 的 `struct` 版本相同；实际上，两个版本的 `p1` 工作方式完全相同。例如，要访问数据，只要使用通用语法：

```
p1.x = 5;
p1.y = 6;
```

指针的工作方式也相同。如果定义一个：

```
Point *p1;
```

然后可以使用 `malloc()` 或 `new` 来为它分配内存：

```
p1=new Point;
```

对 `p1` 进行赋值：

```
p1->x = 5;
```

```
p1->y = 6;
```

最低限度来讲，在访问公有数据元素时类和结构是相同的。关键术语 `public` 表示什么意思呢？再看一下前面的几个 `Point` 类的实例，可以定义：

```
class Point
{
public:
int x, y;
};
```

上述代码中在定义声明之前有关键词：`public`。这样就定义了变量（和成员函数）的作用范围。作用范围选项还有许多，但是通常只使用两种——`public` 和 `private`。

公有和私有

如果在所有的只包含数据的类定义之前使用关键词 `public`，那就只是一个标准结构。换句话说，结构就是具有公有作用范围的类。公有作用范围表示任何人都可以看到该类数据元素。对于主程序、其他函数、成员函数中的代码，数据就不能隐藏或密封。而私有作用范围则禁止不是该类部分的其他函数使用该数据。例如，请看下面的类：

```
class Vector3D
{
public:
int x, y, z; // anyone can mess with these
private:
int reference_count; // this is hidden
};
```

`Vector3D` 分为两个部分：公有数据域和私有数据域。公有数据域具有三个子段：`x`、`y`、`z`，任何人都可以改变它们。另外在私有数据域还有一个隐藏的字段：`reference_count`。该字段对所有的函数都是隐藏的，该类的成员函数（也不是任何类都可以的）除外。因此，如果编写了像下面这样的代码：

```
Vector3D
v.reference_count = 1; // illegal!
```

编译器将会提示程序出错！因此就存在这样的问题，如果不能访问私有变量的话，这些私有变量有什么好呢？实际上，这些私有变量对于编写一些不希望或不需要用户改变的内部工作变量的黑匣子式的类是非常重要的。在这种情况下，就必须使用私有作用范围。但是，要访问私有成员的话，就必须为该类添加成员函数或方法，由此我们学习下一部分内容。

类成员函数 (A. K. A. 方法)

成员函数，即方法是在一个类中并且仅在此类中运行的一个函数。实例如下：

```
class Vector3D
{
public:
    int x, y, z; // anyone can mess with these
    // this is a member function
    int length(void)
    {
        return(sqrt(x*x+y*y+z*z));
    } // end length;
private:
    int reference_count; // this is hidden
};
```

注意黑体显示的成员函数 `length()`。我已经在该类中定义了一个函数！感到古怪吗？看下面如何使用该函数

```
vector3D v; // create a vector
// set the values
v.x = 1;
v.y = 2;
v.z = 3;
// here is the cool part
printf("\nlength = %d", v.length());
```

也可以像访问一个元素一样调用一个类成员函数。如果 `v` 是一个指针的话，可以：

```
v->length();
```

现在，你可能会说：“我有大约 100 个必须访问该类数据的函数；但是我不可能将它们都放在这个类中！”实际上，只要愿意，可以将这 100 个函数都放在一个类中，但是我认为那样可能会造成混乱。可以在类定义之前定义类成员函数。我们稍后再讨论一下类成员函

数。下面我添加另外一个成员函数，示范一下如何访问私有数据成员 `reference_count`:

```
class Vector3D
{
public:
    int x, y, z; // anyone can mess with these
    // this is a member function
    int length(void)
    {
        return(sqrt(x*x+y*y+z*z));
    } // end length;
    // data access member function
    void addref(void)
    {
        // this function increments the reference count
        reference_count++;
    } //end addref;
private:
    int reference_count; // this is hidden
};
```

通过成员函数 `addref()` 可以和 `reference_count` 进行交流。这种做法看上去是多余的，但是如果仔细考虑一下的话，这的确是个很好的办法。现在用户就不能对该数据成员做任何愚蠢的举动了。这样做将一直允许访问函数，这种情况允许调用程序使 `reference_count` 加 1:

```
v.addref();
```

该调用程序不能改变参考计数，也不能乘以一个数等等，这是因为 `reference_count` 是私有的。只有该类的成员函数能够访问该调用程序，这就是数据隐藏和禁止。

在这一点上，我认为你正在看到了类的功能。可以使用类数据的结构来填充类，在操作该数据的类中添加函数，隐藏数据等等，太完美了，当然还可以做得更好。

构造函数和解构函数

如果你已经使用 C 语言编程了一个星期以上，我相信你已经做了上百万次的初始化一个结构的工作。例如说，要创建一个结构 `Person`:

```
struct Person
{
    int age;
    char *address;
    int salary;
};
Person people[1000];
```

下面就要初始化 1000 个人的结构。应当:


```
for (int index = 0; index<1000; index++)
{
    people[index].age = 18;
    people[index].address = NULL;
    people[index].salary = 35000;
} // end for index
```

但是如果忘记了初始化结构怎么办，那如何使用该结构？那就要察看常规预防故障手册。同样在程序运行过程中，如何来分配内存，并为一个地址段指定内存？

```
person[20].address = malloc(1000);
```

然后使用该内存，如果忘记了该内存，再按下面做法进行：

```
person[20].address = malloc(4000);
```

这样的话，一千字节的内存就被占用了。在分配内存之前，应当通过调用 `free()` 函数来释放内存：

```
free(people[20].address);
```

我想你可能已经这样做了。C++ 通过在创建类时调用两个新的自动函数来解决这些内部处理问题：构造函数和解构函数。

当使用具体实例说明一个类对象时调用构造函数。例如，执行下面程序时：

```
vector3D v;
```

调用默认的构造函数，此时构造函数没有任何动作。同样，当 `v` 超出作用范围时，也就是说，当定义 `v` 的函数终止时；或者如果 `v` 是一个全局变量，当程序终止时，调用默认的解构函数，该函数也没有任何动作。要清楚具体操作，应当编写一个构造函数和解构函数。如果不了解它们的具体操作，就不必编写该函数了，可以定义其中的一个或两个函数。

构造函数的编写

让我们使用 `person` 结构转换为一个类来作为一个实例：

```
class Person
{
public:
    int age;
    char *address;
    int salary;
    // this is the default constructor
    // constructors can take a void, or any other set of parms
    // but they never return anything, not even a void
```

```

Person()
{
    age    =0;
    adress = NULL;
    salary =35000;
} // end person
}

```

注意，构造函数和类具有相同的名称，这里指的是 `Person`。这并不是巧合，而是个规律。另外，构造函数不返回任何值，必须这样。但是构造函数可以携带参数。在这里，它不携带任何参数，但是可以创建有参数的构造函数。实际上能够建立无限多个不同的构造函数，每个函数具有不同的调用清单。这是如何使用不同的调用创建各种类型的 `Person`。总之要创建一个 `Person`，并且能够自动初始化，可以：

```
Person person1;
```

这样将自动调用构造函数，并依次进行下面分配：

```

people[index].age    =0;
people[index].adress = NULL;
people[index].salary =35000;

```

当程序如下编写时，构造函数发挥作用：

```
Person person[1000];
```

每个 `Person` 的实例都要调用构造函数，并将初始化所有的 1000 个人而无需另外代码。

下面我们讨论一下更高级的内容。请想一想，函数如何重载？同样也可以重载构造函数。因此，如果想创建一个依次设置为年龄、地址和薪水的构造函数，应当：

```

class Person
{
public:
    int age;
    char *address;
    int salary;
    // this is the default constructor
    // constructors can take a void, or any other set of parms
    // but they never return anything, not even a void
    Person()
    {
        age    =0; adress = NULL; salary =35000;
    } // end person
    // here is our new more powerful constructor
    person(int new_age, char *new_address, int new_salary)
    {
        // set the age

```

```

age= new_age;
// allocate the memory for the address and set address
address = new char[string(new_address)+1];
strcpy(address, new_address);
//set salary
salary = new_salary;
} // end Person int, char *, int
};

```

现在就有了两个构造函数，一个没有参数，一个有三个参数：一个是整数，一个是字符串，另一个也是整数。下面是创建一个 24 岁、居住在枫树大街 500 号、年薪 52000 美元的人的实例：

```
Person person2(24, "500 Maple Street", 52000);
```

当然，可能你认为可以使用不同的语法来初始化一个 C 结构，如：

```
Person person = {24, "500 Maple Street", 52000};
```

但是，如何进行内存分配？如何进行字符串复制等等？标准 C 只能进行盲目的复制，但 C++ 在创建了对象之后，还可以进行代码和逻辑的运行。这样给出了更多的控制。

解构函数的编写

创建了一个对象之后，在某个时刻还要结束该对象。下面是使用 C 语言编写的普通的调用的一个清除函数，但是使用 C++ 可以通过调用解构函数将该对象自动删除。编写一个解构函数要比编写构造函数更简单，因为使用解构函数没什么灵活性，它们只有一种格式：

```
~classname();
```

没有参数，也没有返回类型——周期。没有任何例外！下面在 `Person` 类中添加一个解构函数：

```

class Person
{
public:
    int age;
    char *address;
    int salary;
    // this is the default constructor
    // constructors can take a void, or any other set of parms
    // but they never return anything, not even a void
    Person()
    {
        age = 0;
        adress = NULL;
        salary = 35000;
    } // end person

```

```

/* here is our new more powerful constructor
person(int new_age, char *new_address, int new_salary)
{
    // set the age
    age= new_age;
    // allocate the memory for the address and set address
    address = new char[string(new_address)+1];
    strcpy(address, new_address);
    //set salary
    salary = new_salary;
} // end Person int, char *, int
// here is our destructor
~Person()
{
    free(address);
} // end ~Person
};

```

我已经用黑体显示了解构函数。注意，解构函数中并没有什么特殊的代码；我可以做任何想做的事情。使用这个新的解构函数，就不必操心内存的释放。例如，使用 C 语言在结构中创建一个有内部指针的结构，然后不释放该解构的内存就结束该结构，该内存就永远丢失，这就是内存流失，如下面 C 程序所示。

```

struct
{
    char *name;
    char *ext;
} filename;
foo
{
    filename file; // here is a filename
    filename.name = malloc(80);
    filename.ext = malloc(4);
} // end foo

```

该结构文件删除之后，分配的 84 个字节将永远丢失！但是使用 C++ 程序和解构函数，就不会发生这种情况，因为编译器将自动调用释放内存的解构函数。

上述内容就是关于构造函数和解构函数的基本内容，当然这两个函数的内容还有很多。还有一些特殊的构造函数，如复制构造函数、分配构造函数等等。但对于初学者，这已足够了。而对于解构函数，只有上面所述的一种类型，所以还是比较容易记住的。

作用域操作符

C++中还有一个新的操作符，称之为作用域操作符，用双冒号 (::) 来表示。常用于参考类函数和该类范围中的数据成员。不必对此操作符的含义过于担心；下面就讨论如何使用该操作符在一个类外面定义类函数。

至此已经定义了类定义内部的类成员函数。尽管定义类内部的成员函数对于小型类很适应，但是在大型类中使用则存在一些问题。而如果合理地定义成员函数，并且通知编译器这些函数是类函数，而不是一般的文件一级的函数，就可以在一个类外面任意编写类成员函数。一般来说使用作用域操作符以及下面的语法来实现这一点。

```
return_type class_name::function_name(param_list)
{
    // function body
}
```

当然在类内部，仍然要使用原型来定义该函数（当然要去掉作用域操作符和类名称），但是也可以直到后面再脱离该程序块。下面以 **Person** 类为例，看一下如何使用作用域操作符。下面是去掉函数体的新类：

```
class Person
{
public:
    int age;
    char *address;
    int salary;
    // this is the default constructor
    Person()
    // here is our new more powerful constructor
    person(int new_age, char *new_address, int new_salary)
    // here is our destructor
    ~Person()
};
```

下面是程序体，将它们和其他的函数一起都放在类定义之后：

```
Person::Person()
{
    // this is the default constructor
    // constructors can take a void, or any other set of parms
    // but they never return anything, not even a void
    age = 0;
    adress = NULL;
    salary = 35000;
```

```

} // end person
////////////////////////////////////
Person::Person(int new_age, char *new_address, int new_salary)
{
// here is our new more powerful constructor
// set the age
age= new_age;
// allocate the memory for the address and set address
address = new char[string(new_address)+1];
strcpy(address, new address);
//set salary
salary = new_salary;
} // end Person int, char *, int
////////////////////////////////////
Person::~Person()
{
// here is our destructor
free(address);
} // end -Person

```

提示

大多数程序员喜欢在类名称之前放一个大写的 C。我也经常这样做，我希望你继续这样做。如果我编程时，我可能是使用 Cperson，而不是 Person，或者是使用全部大写 CPERSON 也可以。

函数和操作符重载

我们讨论的最后一个内容是重载，主要分为两部分：函数重载和操作符重载。现在没有时间来详细解释操作符重载，只以一些常用实例进行解释。假定有一个 Vector3D 类，想使两个矢量相加 $v1+v2$ ，并将结果保存于 $v3$ 中。可以如下这样做：

```

Vector3D v1 = {1, 3, 5},
          v2 = {5, 9, 8},
          v3 = {0, 0, 0};
// define an addition function, this could have
// been a class function
Vector3D Vector3D_Add(Vector3D v1, Vector3D, v2)
{
Vector3D sum; // temporary used to hold sum
sum.x = v1.x+v2.x;
sum.y = v1.y+v2.y;
sum.z = v1.z+v2.z;
return(sum);
} // end Vector3D_Add

```

然后，要将该矢量添加到该函数中，应当编写如下代码：

```
v3 = Vector3D::Add(v1, v2);
```

虽然很粗糙，但也能运行。使用 C++ 和操作符重载，可以重载+操作符，并建立一个新版本的+操作符，添加到矢量中。因此：

```
v3=v1+v2;
```

简单吧！下面是重载操作符函数的语法，应当阅读一下有关 C++ 内容的书籍来了解其细节：

```
class Vector3D
{
public:
    int x, y, z; // anyone can mess with these
    // this is a member function
    int length(void) {return(sqrt(x*x+y*y+z*z));}
    // overloaded the + operator
    Vector3D operator+( Vector3D &v2)
    {
        Vector3D sum; // temporary used to hold sum
        sum.x = v1.x+v2.x;
        sum.y = v1.y+v2.y;
        sum.z = v1.z+v2.z;
        return(sum);
    }
private:
    int reference_count; // this is hidden
};
```

注意，第一个参数是隐含的对象，所以参数表中只有 v2。总之，操作符重载功能非常强大。使用它，可以创建新的数据类型和操作符，以便于不使用函数调用就可以完成各种很酷的操作过程。

函数重载在我们讨论构造函数时就进行了讨论。函数重载知识使用两个或多个具有相同名字、不同参数的函数。假如说我们编写一个称为 plot pixel 的函数，具有下述功能：如果不使用参数而调用该函数，就在当前光标位置上绘制一个点，如果使用参数 x、y 来调用该函数，则在 x、y 位置上绘制一个点。如下所示：

```
int cursor_x, cursor_y; // global cursor position
// the first version of Plot_Pixel
void Plot_Pixel(void)
{
    // plot a pixel at the cursor position
    plot(cursor_x, cursor_y);
}
```

```
////////////////////////////////////  
// the second version of Plot_Pixel  
void Plot_Pixel(int x, int y)  
{  
    // plot a pixel at the sent position and update  
    //cursor  
    plot(cursor_x=x, cursor_y=y);  
}
```

可以如下调用函数:

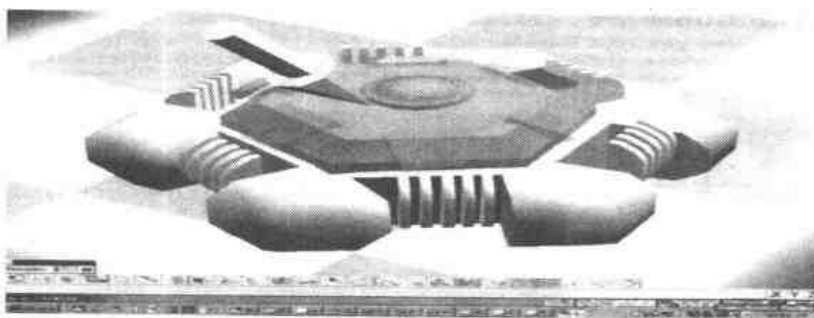
```
Plot_Pixel(10, 10); // calls version 2  
Plot_Pixel(); // calls version 1
```

提示

编译器知道该两个函数的区别, 因为创建真实的函数名不仅要使用函数名, 而且还要使用参数列表的不同版本, 在编译器的名称空间中创建一个统一的名称。

小结

上面我们简单浏览了一下 C++ 语言的一些内容。如果 Robert Lafore (世界上最优秀的 C++ 作者) 阅读了该内容, 他会因为我如此不严密面杀了我的。但是不管怎么讲, 现在你可能了解了一些该语言的知识, 即使不编写 C++ 程序, 至少也能够阅读 C++ 程序。



E

游戏编程资源

下面是一些资源的杂录，可能对你成为一个游戏程序员有一些帮助。

游戏编程网站

重要的游戏编程网站有上百个，不能全部罗列。下面是我感兴趣的几个网站：

GameDev.Net

<http://www.gamedev.net/>

The Game Programming Megasite

<http://www.perplexed.com>

The Official MAME Page

<http://mame.retrogames.com/>

The Games Domain

<http://www.gamesdomain.com/>

Top 50 Game Programming Sites

<http://qbrpgt50.hypermart.net/top50/topsite.html>

The Coding Nexus

<http://www.gamesdomain.com/gamedev/gprog.html>

The Computer Game Developer's Conference

<http://cgdc.com>

下载站点

游戏程序员需要优秀的游戏、工具、应用和原始资料。下面列出了我喜欢下载内容的站点：

Happy Puppy	http://www.happypuppy.com
Game Pen	http://www.gamepen.com/topten.asp
Ziff Davis Net	http://www.zdnet.com/swlib/game.html
Adrenaline Vault	http://www.avault.com/pcrl
Download.Com	http://www.download.com/pc/cdoor/0,323,0-17,00.html?st.d1.fd.cats.cat17
Jumbo.Com	http://www.jumbo.com/pages/games/windows95/games2/
GT Interactive	http://www.gtgames.com
Apogee	http://www.apogee1.com
Epic Megagames	http://www.epicgames.com
CNet	http://www.cnet.com
WinFiles.com	http://www.winfiles.com
eGames	http://www.egames.com

2D/3D 引擎

网站上的一个站点集中了所有的 3D 开发引擎。该站点是 3D 引擎列表，包含了使用不同技术的 3D 引擎。令人激动的是有许多可以免费使用的引擎！下面是其地址：

<http://cg.cs.tu-berlin.de/~ki/engines.html>

另外还有一些其他的专用 2D/3D 引擎的连接：

Genesis 3D Engine	http://www.genesis3d.com
DigitalFX Engine	http://www.fastprojects.com
SciTech MGL	http://www.scitechsoft.com
Crystal Space	http://crystal.linuxgames.com

游戏编程书籍

关于图像、声音、多媒体和游戏开发的书籍有很多，但是全部购买就太昂贵了。下面列出了一些可以浏览和游戏相关的书籍并且可以下载的一些站点：

Games Domain Bookstore

<http://www.gamesdomain.com/gamedev/gdevbook.html>

Opifex Bookstore

http://www.opifex.freemove.co.uk/bookstore_com.html

Programmer's Vault

http://www.chesworth.com/pv/vault/bookshop/game_programming.htm

Microsoft DirectX 多媒体展示

毋庸置疑，Microsoft 是世界上最大的网站，有上千个网页、网区和 FTP 站点等等。但是你可能感兴趣的网站也就是 DirectX Multi Expo:

<http://www.microsoft.com/directx/>

在该网页上，可以看到最新的新闻，可以下载最新版本的 DirectX、DirectMedia 以及老版本的修补程序。你至少应当每周花一个小时来阅读这些信息。这样可以使你能够一直跟上 Microsoft 和 DirectX 的发展的脚步。

世界性新闻组网络系统

我从来不关心 Internet 新闻组，因为联系起来速度非常缓慢。但也有些有价值的新闻组织的浏览：

alt.games

rec.games.programmer

comp.graphics.algorithms

comp.graphics.animation

comp.ai.games

如果以前没有阅读过新闻组，请浏览一下……你需要一个新闻阅读器，以便于能够下载信息以及阅读信息线程。大多数网络浏览器，如 Netscape Navigator 和 Internet Explorer 都有内置的新闻阅读器。只要阅读帮助文件，指出如何建立阅读新闻组的浏览器，然后登录一个新闻组，例如 alt.games，下载所有的信息，然后开始阅读。

抓住产业：蓝调新闻

Internet 带宽大约 99.9%都浪费了。大部分是由于许多人在反反复复地胡说，另外还充满了荒诞的联络。但也有一些网站不会浪费你的时间。其中就有蓝调新闻，它是各种行业

要人发表他们当天想法的地方。在下面网站即可登录：

<http://www.bluenews.com>

每天都可以察看一下内容。

游戏开发杂志

据我所知，只有两种英文游戏开发杂志。第一种也是最大的一种是游戏开发者，月刊，内容有游戏编程、艺术、3D 建模、市场行情等等。其网址是：

<http://www.gdmag.com>

也可以访问其姊妹站点 Gamasutra：

<http://www.gamasutra.com>

第二种游戏开发杂志是 The Cursor，它是一种具有更多的自由格式的基础类杂志。其网址是：

<http://www.thecursor.com>

游戏网站开发人员

创建一个游戏时应当考虑的最后一件事情是该游戏的网站。如果正在推销一种作为共享软件的游戏，建立一个很小的站点来展示该游戏是很重要的。应当了解如何使用 FrontPage 或 Netscape 中的简单的网站编辑器，但是如果建立一个很酷的网站来展示你编写的游戏并且使其更具有吸引力，就应当建立专业网站。我在一些非常糟糕的网站上看到过一些非常好的游戏，每每都觉得十分可惜。

我使用的公司名称是 Belm Design Group。它们可以帮助你建立一个自己的游戏网站，通常要 500.00~3000.00 美金，网址是：

<http://www.belmdesigngroup.com>

Xtreme Games LLC

我自己的公司是 Xtreme Games LLC。我们开发并出版 2D/3D 的 PC 平台的游戏。我们的站点是：

<http://www.xgames3d.com>

在网站内容有 3D 图像、人工智能、物理、DirectX 以及其他内容等。另外，我将通知

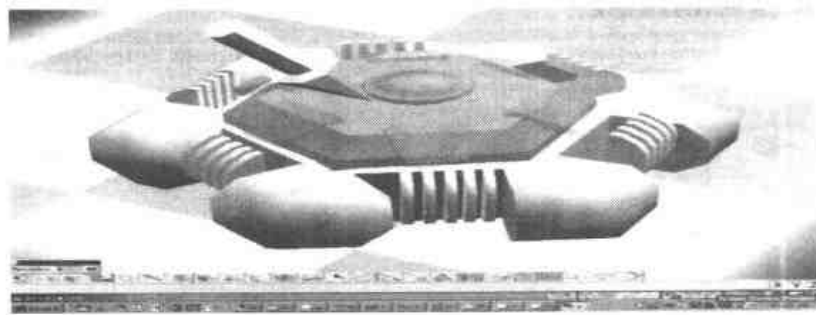
本书的改变和新加的内容。

Xtreme Games LLC 出版并开发游戏。如果你有了很好的游戏，可以通过 Xtreme 登录并检索游戏作者的信息。我们也为开发人员提供技术支持。

下面是我的电子信箱：

CEO@xgames3d.com

E



ASCII 表

ASCII 表是长时期以来我一直追寻的东西。我想大概只有 Peter Norton 的计算机书中有 ASCII 表。其实每本计算机书都应该附有 ASCII 表，不过我自己过去的书中也没有。但我已改正了这个错误。看，这里就是字符 0~127，127~255 的完全注释的 ASCII 表。

十进制	十六进制	ASCII	十进制	十六进制	ASCII
000	00	空	027	1B	←
001	01	☺	028	1C	L
002	02	☻	029	1D	↔
003	03	♥	030	1E	▲
004	04	♦	031	1F	▼
005	05	♣	032	20	空格
006	06	♠	033	21	!
007	07	•	034	22	"
008	08	█	035	23	#
009	09	○	036	24	\$
010	0A	◼	037	25	%
011	0B	♂	038	26	&
012	0C	♀	039	27	'
013	0D	♪	040	28	(
014	0E	♫	041	29)
015	0F	⊗	042	2A	*
016	10	•	043	2B	+
017	11	◀	044	2C	,
018	12	↓	045	2D	-
019	13	!!	046	2E	.
020	14	¶	047	2F	/
021	15	§	048	30	0
022	16	—	049	31	1
023	17	↓	050	32	2
024	18	↑	051	33	3
025	19	↓	052	34	4
026	1A	→	053	35	5

续表

十进制	十六进制	ASCII	十进制	十六进制	ASCII
054	36	6	081	51	Q
055	37	7	082	52	R
056	38	8	083	53	S
057	39	9	084	54	T
058	3A	:	085	55	U
059	3B	;	086	56	V
060	3C	<	087	57	W
061	3D	=	088	58	X
062	3E	>	089	59	Y
063	3F	?	090	5A	Z
064	40	@	091	5B	[
065	41	A	092	5C	\
066	42	B	093	5D]
067	43	C	094	5E	^
068	44	D	095	5F	_
069	45	E	096	60	`
070	46	F	097	61	a
071	47	G	098	62	b
072	48	H	099	63	c
073	49	I	100	64	d
074	4A	J	101	65	e
075	4B	K	102	66	f
076	4C	L	103	67	g
077	4D	M	104	68	h
078	4E	N	105	69	i
079	4F	O	106	6A	j
080	50	P	107	6B	k

续表

十进制	十六进制	ASCII	十进制	十六进制	ASCII
108	6C	l	135	87	ç
109	6D	m	136	88	ê
110	6E	n	137	89	ë
111	6F	o	138	8A	è
112	70	p	139	8B	ï
113	71	q	140	8C	î
114	72	r	141	8D	ì
115	73	s	142	8E	ä
116	74	t	143	8F	å
117	75	u	144	90	é
118	76	v	145	91	æ
119	77	w	146	92	œ
120	78	x	147	93	ô
121	79	y	148	94	ö
122	7A	z	149	95	ò
123	7B	{	150	96	û
124	7C		151	97	ù
125	7D	}	152	98	ÿ
126	7E	~	153	99	õ
127	7F	Δ	154	9A	ú
128	80	ç	155	9B	ƒ
129	81	Û	156	9C	£
130	82	é	157	9D	¥
131	83	â	158	9E	P _t
132	84	ä	159	9F	f
133	85	à	160	A0	á
134	86	ã	161	A1	í

续表

十进制	十六进制	ASCII	十进制	十六进制	ASCII
162	A2	ó	189	BD	ſ
163	A3	ú	190	BE	ſ
164	A4	ñ	191	BF	ſ
165	A5	Ñ	192	C0	Ł
166	A6	<u>a</u>	193	C1	Ł
167	A7	<u>o</u>	194	C2	Ł
168	A8	¿	195	C3	Ł
169	A9	¬	196	C4	Ł
170	AA	¬	197	C5	Ł
171	AB	½	198	C6	Ł
172	AC	¼	199	C7	Ł
173	AD	¡	200	C8	Ł
174	AE	«	201	C9	Ł
175	AF	»	202	CA	Ł
176	B0	■	203	CB	Ł
177	B1	■	204	CC	Ł
178	B2	■	205	CD	Ł
179	B3		206	CE	Ł
180	B4	†	207	CF	Ł
181	B5	‡	208	D0	Ł
182	B6	‡	209	D1	Ł
183	B7	¶	210	D2	Ł
184	B8	¶	211	D3	Ł
185	B9	‡	212	D4	Ł
186	BA	¶	213	D5	Ł
187	BB	¶	214	D6	Ł
188	BC	Ł	215	D7	Ł

续表

十进制	十六进制	ASCII	十进制	十六进制	ASCII
216	D8	±	241	F1	±
217	D9	┘	242	F2	≥
218	DA	┐	243	F3	≤
219	DB	■	244	F4	┌
220	DC	■	245	F5	└
221	DD	└	246	F6	÷
222	DE	└	247	F7	≈
223	DF	~	248	F8	°
224	E0	α	249	F9	•
225	E1	β	250	FA	.
226	E2	γ	251	FB	√
227	E3	π	252	FC	N
228	E4	Σ	253	FD	2
229	E5	σ	254	FE	■
230	E6	μ	255	FF	
231	E7	γ			
232	E8	φ			
233	E9	θ			
234	EA	Ω			
235	EB	δ			
236	EC	∞			
237	ED	∅			
238	EE	€			
239	EF	∩			
240	F0	≡			